



**SAPIENZA**  
UNIVERSITÀ DI ROMA

## Integrazione di Electronic Health Records per la stratificazione dei pazienti

Facoltà di ingegneria dell'informazione, informatica e statistica  
Laurea Triennale in Informatica

**Daniele Silvestri**  
Matricola 1799313

A handwritten signature in black ink that reads 'Daniele Silvestri'.

Relatore  
Prof. Enrico Tronci

A handwritten signature in black ink that reads 'Enrico Tronci'.

Tesi non ancora discussa

---

**Integrazione di Electronic Health Records per la stratificazione dei pazienti**

Relazione di Tirocinio. Sapienza Università di Roma

© 2022 Daniele Silvestri. Tutti i diritti riservati

Questa tesi è stata composta con L<sup>A</sup>T<sub>E</sub>X e la classe Sapthesis.

Email dell'autore: [silvestri.1799313@studenti.uniroma1.it](mailto:silvestri.1799313@studenti.uniroma1.it)

## Sommario

### 0.1 Contesto

Ci troviamo nel mondo della sanità,più precisamente nelle strutture ospedaliere, dove ogni giorno entrano tantissime persone,le quali, hanno bisogno di una diagnosi tempestiva che possa porre rimedio ai loro problemi di salute. In molti casi questa diagnosi può essere immediata mentre in altri può occupare un pò di tempo in più,in ogni caso la tempestività nell'effettuare una diagnosi e somministrare i medicinali giusti al paziente potrebbe essere decisiva per il suo destino.

### 0.2 Motivazioni

In questo contesto potrebbe essere d'aiuto lo sviluppo di algoritmi,basati sulla raccolta dati degli ospedali,che possono aiutarci sia per quanto riguarda la ricerca della cura più efficace per ogni paziente sia per una predizione basata sui dati vitali del paziente che possa rendere più chiara la sua situazione clinica.

Inoltre,grazie alla raccolta dei dati e agli algoritmi potremmo puntare ad avere ovunque una sanità "coerente" che indipendentemente dalla posizione geografica e dall'afflusso di pazienti abbia gli stessi dati e metodi di tutti gli altri.

### 0.3 Contributi

Il nostro contributo lo daremo attraverso l'organizzazione,l'elaborazione dei dati raccolti e tramite lo sviluppo di un predittore,che prendendo in input i dati vitali di un paziente appena entrato in pronto soccorso,possa predire se quest'ultimo sia destinato,oppure no,nel reparto di terapia intensiva attraverso l'utilizzo di un albero decisionale.

Tutto questo darà l'idea di come l'informatica e le sue tecnologie nel contesto ospedaliero possano essere potenzialmente estremamente utili e importanti per sviluppare sempre più cure efficaci e portare l'efficienza della sanità a livelli sempre più alti.



# Indice

0.1 Contesto	iii
0.2 Motivazioni	iii
0.3 Contributi	iii
<b>1 Introduzione</b>	<b>1</b>
1.1 Contesto	1
1.2 Motivazioni	1
1.3 Contributi	2
1.4 Lavori correlati	2
1.5 Outline	3
<b>2 Background</b>	<b>5</b>
2.1 I progetti MIMIC IV e MIMIC ED	5
2.1.1 MIMIC-IV	5
2.1.2 MIMIC-ED	6
2.2 Struttura di un nodo in NEO4J	6
<b>3 Metodi</b>	<b>9</b>
3.1 Trasformazione in grafo(funzione csvToNeo(path))	9
3.2 Predittore(funzione predittore(path))	10
<b>4 Implementazione</b>	<b>11</b>
4.1 Implementazione di csvToNeo(path)	11
4.1.1 Primo step:scorrimento e lettura dei file .csv(tabelle)	11
4.1.2 Secondo step:ricerca dei campi presenti in più di una tabella e collegamento alle tabelle a cui appartengono nel grafo	13
4.1.3 Terzo step:Creazione del grafo	13
4.1.4 Quarto step:traduzione del grafo in QUERY Cypher	14
4.2 Implementazione di predittore(path)	15
4.2.1 Raccolta dei dati dalle diverse tabelle	15
4.2.2 Mapping dei ricoveri e raccolta dati pazienti in terapia intensiva	15
4.2.3 Assegnazione valore al campo icu e creazione del file csv	16
4.2.4 Descrizione della tabella result.csv	17
<b>5 Risultati degli esperimenti della trasformazione in grafo</b>	<b>19</b>
5.1 Obiettivi	19
5.2 Caricamento e impostazione della query in NEO 4J	20
5.3 Correttezza	22
5.3.1 Verifica struttura MIMIC IV	22
5.3.2 Verifica di "Core"	23
5.3.3 Verifica di "Hosp"	24
5.3.4 Verifica di "Icu"	27
5.3.5 Verifica struttura di MIMIC-ED	28
5.3.6 Verifica sui campi ripetuti	30

5.3.7	Verifica collegamento tra i MIMIC IV e MIMIC-ED	32
5.4	Valutazioni Tecniche	34
5.4.1	Numero totale dei nodi	34
5.4.2	Numero totale delle relazioni	35
5.4.3	Unicità delle proprietà	36
5.4.4	Il percorso più corto tra due nodi	37
<b>6</b>	<b>Risultati degli esperimenti sul predittore</b>	<b>39</b>
6.1	Obiettivi	39
6.2	Settaggi	39
6.2.1	Primo passaggio: conversione dei campi da numerici a nominali	40
6.2.2	Secondo passaggio: bilanciamento delle istanze del campo "icu"	41
6.2.3	Il classificatore J48	42
6.2.4	Settaggi pre-esecuzione J48	43
6.3	Valutazione sulla correttezza del predittore	45
6.3.1	Stampa dell'albero decisionale	47
<b>7</b>	<b>Conclusioni</b>	<b>49</b>

# Capitolo 1

## Introduzione

Digitalizzazione degli ospedali e delle cartelle cliniche, questa è la sfida che l'Unione Europea sta portando avanti da qualche anno e vuole portare a termine nel minor tempo possibile.

Quello che andremo a vedere in questa relazione di tirocinio riguarda l'health-care e da ciò che andremo a fare si terranno spunti su come questa evoluzione potrebbe portare a risvolti molto interessanti, come lo sviluppo di algoritmi sempre più accurati che potrebbero prevedere il destino di un paziente ed aiutare i medici nella scelta della cura più adatta per ogni paziente. Per arrivare ai nostri obiettivi, andremo ad usufruire e a manipolare i dati dei ricoveri raccolti direttamente in un ospedale presenti nei progetti MIMIC-IV<sup>[1]</sup> e MIMIC-ED<sup>[2]</sup> di PhysioNet.

### 1.1 Contesto

Ci troviamo, come già detto, nel contesto ospedaliero dove i medici ogni giorno trovano davanti i loro occhi tantissimi pazienti tutti con, molto spesso, diversi problemi ai quali bisogna dare una diagnosi adeguata e più rapida possibile per cercare di preservarne la salute.

Proprio a causa di ciò, un medico potrebbe trovare utile una soluzione che lo aiuti innanzitutto nella diagnosi del paziente e che, magari, lo guidi verso la giusta strada da intraprendere per trovare la causa dei problemi del malato e successivamente sostenerlo nel suggerirgli una cura che possa essere efficiente in quel caso specifico.

In tutto questo l'aiuto di un algoritmo, o anche semplicemente l'avere a disposizione dati di pazienti con simili caratteristiche, potrebbe rivelarsi estremamente importante per ridurre al minimo i tempi di diagnosi in un contesto come questo dove anche pochi secondi possono fare la differenza.

### 1.2 Motivazioni

Al giorno d'oggi, fortunatamente, possiamo contare su strutture ospedaliere complete ed efficienti in tutto il territorio italiano ed europeo, ma abbiamo ancora bisogno di fare uno step per alzare maggiormente i livelli di efficienza nel campo della medicina.

Ciò di cui abbiamo bisogno è di rendere le cure uguali in tutti gli ospedali e che tutti i medici, indipendentemente dalla loro posizione geografica, abbiano accesso a tutti i dati di tutte le altre strutture per poterle consultare.

Ovviamente, è di facile intuizione che una struttura ospedaliera, per esempio, nella città di Roma avrà un numero di pazienti molto più grande in confronto ad un ospedale situato in una cittadina di provincia e questo, in caso di arrivo nell'ospedale di un paziente con sintomi poco comuni, potrebbe essere molto significativo. Avendo a disposizione più dati il centro di Roma potrebbe essere maggiormente pronto e tempestivo nella diagnosi e successivamente trovare la cura più adatta

per il paziente. Quello che noi andremo a vedere nel corso di questa relazione è l'esempio di come la raccolta dati e lo sviluppo di algoritmi sui dati possa essere d'aiuto per avvicinarci a questo ambizioso obiettivo.

### 1.3 Contributi

Con la nostra implementazione andiamo a contribuire e a dare l'idea di come sia possibile realizzare qualcosa di interessante tramite l'elaborazione dei dati raccolti, perché non bisogna solo raccoglierci, ma successivamente bisogna trovare il modo di organizzarli adeguatamente così da renderli utili. Come vedremo, il nostro studio si divide fondamentalmente in due parti:

*la prima* è la conversione in grafo da database relazionale e la connessione dei progetti MIMIC-IV e MIMIC-ED;

*la seconda* riguarda la raccolta di specifici dati dei pazienti in entrata al pronto soccorso e lo sviluppo di un predittore su di essi, realizzato grazie all'uso di un Decision Tree, sulla loro futura entrata o non entrata nel reparto di terapia intensiva.

Con la *prima parte*, oltre che ad ottenere un grafo con entrambi i progetti uniti e quindi una struttura più facile da esplorare, ci avviciniamo al modello dei dati comune OMOP, di fatto, il graph DB è lo step di mezzo tra database relazionale e OMOP.

Vediamo cos'è OMOP [3](#).

Il concetto alla base di questo approccio è trasformare i dati contenuti in tali database in un formato comune (modello di dati) nonché in una rappresentazione comune (terminologie, vocabolari, schemi di codifica), e quindi eseguire analisi sistematiche utilizzando una libreria di routine analitiche standard che sono state scritte in base al formato comune.

Tutto ciò non lo vedremo direttamente noi in questa relazione, ma è molto utile conoscerlo per eventuali sviluppi futuri e proprio la trasformazione del database relazionale in un graph db è uno step iniziale necessario per avvicinarsi a questo modello.

Con la *seconda parte* andiamo a dare un'idea di come gli algoritmi possano essere molto utili per l'immediatezza, più nello specifico, nel pronto soccorso. L'implementazione del predittore si avvicina, di fatti, all'idea che tutti i medici abbiano un "metodo comune" da seguire e che semplicemente dalla rilevazione dei segni vitali del paziente si possa immediatamente avere un'immagine di come si potrebbe evolvere la situazione semplicemente visionando i dati raccolti confrontandoli con l'albero decisionale ottenuto del predittore.

### 1.4 Lavori correlati

Sulla trasformazione dei progetti MIMIC-IV e MIMIC-ED da database relazionali a grafo non esistono veri e propri lavori correlati, possiamo però prendere spunto dal seguente progetto ancora in sviluppo [4](#).

Il progetto, in breve, ha come obiettivo la conversione del progetto MIMIC-IV nel modello dati OMOP, lo standard accennato proprio nello scorso paragrafo.

Il tutto è implementato tramite l'utilizzo del linguaggio SQL con la collaborazione del linguaggio Python. La traduzione effettiva del progetto avviene grazie ad una mappatura manuale dei vocabolari di OMOP (i vocabolari sono, in semplici parole, le linee guida per la traduzione dei dati che variano a seconda del significato e delle caratteristiche dei record).

Il nostro, come andremo a vedere nel corso del documento, è un approccio profondamente diverso. Invece di andare verso OMOP che è un processo lungo e dispendioso ci fermiamo allo step "intermedio", la trasformazione in graph DB, con il quale riusciremo comunque ad arrivare ai nostri obiettivi.

Inoltre, questo potrebbe facilitare anche la successiva conversione nel formato OMOP, dato che



un grafo si avvicina molto più alla struttura di ques'ultimo in confronto ad un semplice database relazionale.

Invece, per quanto riguarda il predittore un lavoro correlato è il seguente [5].

Questo progetto si basa sul riordino dei record e sul successivamente dare una predizione su quello che sarà il percorso clinico del paziente in base allo studio dei dati.

Per arrivare all'obiettivo, gli autori del progetto studiano un vero e proprio metodo basato su complicate formule matematiche applicate sui dati contenuti dai record. Ciò che lo differenzia dal nostro predittore, oltre all'approccio, è l'output finale.

Qui in output vengono visualizzati determinati eventi e le probabilità che essi possano accadere a seconda dei dati raccolti in fase diagnostica. Proprio da questo possiamo osservare che i loro obiettivi sono più "generalisti" rispetto ai nostri, il nostro lavoro è più concentrato sulla predizione del fatto che il paziente finisca oppure no in terapia intensiva rendendo, inoltre, i nostri risultati più immediati e facili da leggere.

## 1.5 Outline

Introduciamo la struttura, seguendo l'ordine dei capitoli, della relazione:

-*Background*: Qui troveremo tutte le conoscenze di base per capire cosa è stato fatto.

-*Metodi*: Descrizione di cosa devono fare gli algoritmi implementati.

-*Implementazione*: Descrizione dell'implementazione degli algoritmi e pseudocodice.

-*Risultati degli esperimenti della trasformazione in grafo*:

Descrizione dei risultati degli esperimenti eseguiti sulla trasformazione in Graph DB del database relazionale.

-*Risultati degli esperimenti sul predittore*:

Descrizione dei risultati degli esperimenti eseguiti sul predittore.

-*Conclusioni*: Riassunto di tutto ciò che è stato fatto e possibili sviluppi futuri.



# Capitolo 2

## Background

Prima di addentrarci nei metodi e nelle implementazioni effettuate abbiamo bisogno di introdurre i punti chiave dei progetti MIMIC IV e MIMIC-ED e di capire come sono rappresentati i grafi in NEO 4J.

### 2.1 I progetti MIMIC IV e MIMIC ED

I progetti Medical Information Mart for Intensive Care IV e ED(MIMIC-IV e MIMIC-ED) sono due database relazionali di raccolta dati per quanto riguarda i ricoveri di un ospedale. Come si può intuire dal nome esteso, il progetto MIMIC nasce per raccogliere informazioni sui ricoveri della terapia intensiva, successivamente viene integrata la parte ED che aggiunge i dati del pronto soccorso e rende tutto quanto molto più completo e interessante.

Vediamo ora una piccola descrizione di entrambi i progetti.

#### 2.1.1 MIMIC-IV

MIMIC-IV [1](#) proviene da due sistemi di database ospedalieri del Beth Israel Deaconess Medical Center: un EHR personalizzato per l'intero ospedale e un sistema informativo clinico specifico per la terapia intensiva.

MIMIC-IV è raggruppato in tre moduli:

*-core*: Il modulo principale memorizza le informazioni di tracciamento del paziente necessarie per qualsiasi analisi dei dati utilizzando MIMIC-IV.

Contiene tre tabelle: patients, admissions e transfers .

Queste tabelle forniscono dati demografici per il paziente, un record per ogni ricovero e un record per ogni degenza in reparto all'interno di un ricovero.

*-hosp*: Il modulo hosp contiene i dati derivati dall'interno dell'ospedale.

Queste misurazioni vengono registrate prevalentemente durante la degenza ospedaliera, sebbene alcune tabelle includano anche dati esterni all'ospedale (es. esami di laboratorio ambulatoriali in labevents).

Contiene le seguenti informazioni (tra parentesi i nomi delle tabelle): misurazioni di laboratorio (labevents ,d\_labitems), colture microbiologiche (microbiologyevents,d\_micro), ordini dal fornitore (poe, poe\_detail), somministrazione di farmaci (emar,emar\_detail), prescrizione di farmaci (prescriptions,pharmacy), informazioni sulla fatturazione dell'ospedale (diagnostics\_icd, d\_icd\_diagnoses, procedures\_icd ,d\_icd\_procedures ,hpcsevents, d\_hcpcs , drgcodes) e informazioni relative al servizio (services).

*-icu*: è il modulo dei dati derivanti dalla terapia intensiva.

I dati documentati nel modulo di terapia intensiva (tra parentesi i nomi delle tabelle) includono input endovenosi e fluidi (*inputevents*), output del paziente (*outputevents*), procedure (*procedureevents*), informazioni documentate come data o ora (*datetimeevents*) e altre informazioni registrate (*charevents*). Tutte le tabelle degli eventi contengono il campo *stay\_id* che consente l'identificazione del paziente in terapia intensiva.

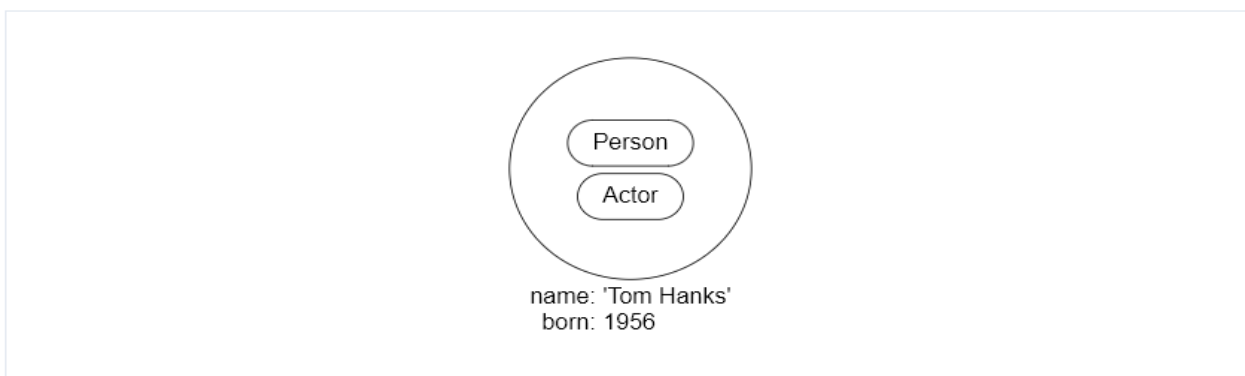
### 2.1.2 MIMIC-ED

MIMIC-ED [2] è un database di ricoveri in pronto soccorso sempre del Beth Israel Deaconess Medical Center. Questo progetto è composto da una singola tabella di monitoraggio del paziente, *edstays* e cinque tabelle di dati: *diagnosis*, *medrecon*, *pyxis*, *trriage* e *vitalsign*.

MIMIC-ED è utilizzabile come database autonomo, ma può anche essere collegato a MIMIC-IV. Il *subject\_id* valore fornisce un collegamento implicito tra i set di dati; ovvero tutti e tre i database si riferiscono allo stesso individuo con lo stesso *subject\_id*. Tutti i ricoveri in MIMIC-ED, rappresentati da *stay\_id*, sono presenti nella tabella *transfers* di MIMIC-IV. I pazienti del pronto soccorso che vengono eventualmente ricoverati nell'unità di terapia intensiva rimangono tracciati e troveremo le informazioni sulla loro successiva permanenza nel modulo di terapia intensiva (modulo *icu*) di MIMIC-IV (tramite la tabella *edstays* e l'identificatore *hadm\_id*). Di conseguenza, MIMIC-ED può essere utilizzato per acquisire informazioni pre-terapia intensiva per pazienti critici in MIMIC-IV, ed è proprio questo che noi andremo a sfruttare per i nostri esperimenti.

## 2.2 Struttura di un nodo in NEO4J

NEO4J sarà il software che useremo per realizzare il nostro Graph DB. Vediamo un esempio per capire come sono strutturati i nodi del grafo. [6]



**Figura 2.1.** Esempio struttura di un nodo in NEO 4J

In questo caso le *label* del nodo sono: *Person* e *Actor*.

Un nodo può avere più di una label. (nel nostro caso come vedremo questo non succederà)

Le **proprietà** sono:

- name:** Tom Hanks
- born:** 1956

Questo nodo può essere creato su NEO4J tramite la seguente QUERY cypher:

```
CREATE (:Person:Actor name: 'Tom Hanks', born: 1956)
```

Come si evince dall'esempio, sostanzialmente il nodo è formato da due componenti: label e proprietà. Possiamo dire, facendo una comparazione con i database relazionali, che la "label" è l'entità a cui l'istanza (ora diventata nodo) appartiene mentre le proprietà sono gli attributi dell'entità stessa.



## Capitolo 3

# Metodi

In questa sezione troveremo la descrizione dei metodi che verranno implementati i quali saranno usati successivamente per effettuare gli esperimenti.

### 3.1 Trasformazione in grafo(funzione csvToNeo(path))

Per quanto riguarda la trasformazione in grafo della base di dati il nostro metodo prenderà in input i due progetti MIMIC-IV e MIMIC-ED, due DB relazionali, e ritornerà in output il Graph DB, più specificatamente il codice in linguaggio CYPHER già pronto per essere caricato nel software NEO 4J.

Per quanto riguarda il processo per la trasformazione da database relazionale a Graph DB sono state seguite le istruzioni del seguente articolo [7](#)

In maniera più dettagliata, il nostro script dovrà quindi inizialmente scorrere tutti i file .csv presenti all'interno delle due cartelle dei progetti tenendo d'occhio e facendo attenzione a ciò che nella trasformazione potrebbe causare problemi per diversi motivi, come per esempio le righe completamente vuote all'interno dei file ed alcuni caratteri che nel linguaggio cypher sono speciali.

Successivamente, il metodo dovrà eseguire gli step che l'articolo, sopra citato, ci indica di seguire:

*-Il primo step* è quello dell'identificazione delle entità, nel nostro caso molto semplice, poichè il nome del file .csv indica il nome della tabella la quale i dati fanno parte.

*-il secondo step* ci indica di dare un nome logico alle relazioni tra le entità per rendere il grafo più leggibile e intuitivo, nel nostro caso però questo passaggio va rivisto poichè non avendo a disposizione un'intelligenza artificiale non è possibile implementarlo.

Quindi, i nomi agli archi che collegano un nodo A con un nodo B sono stati assegnati semplicemente seguendo il formato: "nome nodo A\_nome nodo B".

*-Il terzo e il quarto step* sono la vera e propria trasformazione in grafo.

L'articolo inizialmente ci dice di trasformare tutte le entità in nodi e di far diventare gli attributi delle proprietà.

Successivamente, arriviamo a quello che è effettivamente il quarto ed ultimo step, la trasformazione degli attributi presenti in più di una tabella in nodi. Questi nodi, nella nostra implementazione, saranno collegati ai nodi di "origine" con degli archi partenti dal "nodo proprietà" direzionati verso il nodo originario.

La seguente scelta è stata fatta per risalire più semplicemente e velocemente all'origine dei nuovi nodi derivanti dalle proprietà.

Infine, ci occupiamo di tradurre correttamente il grafo ottenuto nel linguaggio CYPHER e successivamente di scriverlo quindi su un file con estensione ".cypher" pronto per essere eseguito all'interno di NEO 4J.

### 3.2 Predittore(funzione predittore(path))

Il metodo predittore(path) che prepara il file da sottoporre al software Weka prende in input sempre il percorso della cartella contenente i progetti MIMIC e ha come primo obiettivo quello di "raccogliere" i campi che ci interessano per la valutazione dei dati vitali da diverse tabelle presenti nel database, quindi, unirli tutti nel file di output result.csv.

Successivamente, il lavoro che svolge è quello di matchare i dati della tabella "edstays" con la tabella "icustays" (tabella che contiene le informazioni su tutti i ricoveri in terapia intensiva) in modo da andare a creare e assegnare il valore adeguato al campo "icu" che ci indicherà se il paziente ha avuto bisogno della terapia intensiva oppure no.

La tabella "edstays" di MIMIC-ED, in breve, è costruita per fare da "ponte" tra i progetti MIMIC IV e MIMIC-ED e serve per matchare il campo "stay\_id" che è l'identificativo assegnato al ricovero in reparto (quindi quando il paziente per esempio passerà da pronto soccorso a terapia intensiva quest'ultimo cambierà) con il campo "hadm\_id" identificatore che invece rappresenta la degenza ospedaliera e che può essere quindi collegato al progetto MIMIC-IV per ottenere tutti i dati riguardanti il paziente che nel frattempo può aver cambiato diversi reparti.

Dopo questo passaggio, l'ultima cosa che ci rimane da fare è assegnare il valore al campo "icu" aggiunto appositamente per poter effettuare così i nostri test con l'albero decisionale. Il valore del campo icu sarà uguale ad "0" se "hadm\_id" del paziente non è presente all'interno di "icustays", altrimenti se dovesse esserlo il valore sarà uguale ad "1" poichè questo significa che sfortunatamente il paziente ha avuto bisogno della terapia intensiva.



# Capitolo 4

## Implementazione

In questo capitolo vedremo da vicino l'implementazione dei due script i quali sono stati sviluppati in linguaggio Python.

### 4.1 Implementazione di csvToNeo(path)

Iniziamo con la dichiarazione di alcune variabili chiavi dello script:

- relazioni*: dizionario che ha come chiavi i campi, come valori tutte le tabelle dove compaiono.
- tabelle*: dizionario che ha come chiavi i nomi delle tabelle e come valori i campi appartenenti.
- nodi*: stringa dove verrà scritta la query in linguaggio Cypher.

#### 4.1.1 Primo step: scorrimento e lettura dei file .csv(tabelle)

Come primo step per implementare la trasformazione da database relazionale a Graph DB ho deciso di utilizzare la funzione `os.walk()` della libreria `os` [8], grazie alla quale è stato possibile poter scorrere tutti i file e le cartelle presenti all'interno dei progetti MIMIC.

Se l'estensione del file è di tipo ".csv" allora sappiamo che ci interessa e lo apriamo in lettura.

A questo punto, poichè la traduzione da database relazione a grafo per i nostri scopi ci interessa maggiormente da un punto di vista della struttura del progetto, possiamo permetterci di non avere tutti i dati rappresentati al suo interno, quindi ho deciso appositamente di integrare un contatore (contarighe) che ci servirà per limitare il numero di istanze (nel nostro caso 30) per ogni tabella. Questa decisione è stata presa a causa della grande quantità di dati presenti nelle tabelle (alcune arrivano a 500 mila istanze) che gravavano sui tempi di esecuzione dell'algoritmo e sui tempi dell'inserimento del file Cypher in NEO4J.

A questo punto iniziamo a scorrere le varie righe all'interno del file .csv e a comporre la variabile *nodi* iniziando a scrivere al suo interno la query Cypher stando attenti ad alcuni caratteri che potrebbero provocare poi errori durante l'esecuzione della query all'interno di Neo4J. Vediamo quindi lo pseudocodice della lettura dei file e del controllo sui caratteri "fastidiosi":

```

for riga in file:
    c=0 #contatore per le colonne
    contarighe+= 1 #contatore per le righe
    nodi+="CREATE(" + "n:" + nome_tabella + "{"
    for campo in campi:
        se "" in riga[c]: #casi che vanno corretti per evitare errore in query cypher(il carattere " ' " in cypher è speciale e "\" serve per renderlo "annullarlo")
            allora sostituisci il carattere " ' " con " \' "
        se "\\" in riga[c] e la lunghezza di riga[c]== 1:
            allora sostituisci il carattere "\\" con "\\\\"
        ....
        se c < lunghezza di riga -1 :
            allora nodi+= campo + ":" + "" + righe[c] + ","
        else: #l'ultima proprietà della tabella non ha bisogno delle "," alla fine
            nodi+= campo + ":" + "" + righe[c] + "" #per ultimo attributo la virgola non serve
        ....
    c+=1
    nodi+="}}\\n\\nWITH 1 as dummy \\n"
    se contarighe > 29: #per il limite impostato a 30 istanze per ogni file tabella
        break

```

**Figura 4.1.** Pseudocodice

Successivamente effettuiamo il controllo per identificare i campi chiave che per struttura del database sono tutti quanti i campi che terminano con "\_id" più tre campi i quali fanno eccezione e sono le chiavi:code,icd\_code e itemid. Anche questo è un passaggio importante poichè i campi che sono chiave di una tabella non vanno trasformati in nodi ma rimangono semplicemente proprietà nel passaggio in Graph DB. Ciò ci serve anche per iniziare a popolare il dizionario *relazioni* per iniziare a costruire e a trovare gli archi del grafo che corrispondono ovviamente alle relazioni del db relazionale. Vediamo quindi tramite pseudocodice l'implementazione di tale passaggio.

```

for campo in campi:
    se campo[len(campo)-3:] == "_id" oppure campo == "code" oppure campo=="icd_code" oppure campo=="itemid":
        se campo non in relazioni:
            allora relazioni[campo]=[nome_tabella]
        else:
            allora relazioni[campo].append(nome_tabella)

```

**Figura 4.2.** Pseudocodice

Infine, prima di passare alla tabella successiva inseriamo tutti i campi nel dizionario *tabelle* con chiave il nome della tabella a cui appartengono.

#### 4.1.2 Secondo step:ricerca dei campi presenti in più di una tabella e collegamento alle tabelle a cui appartengono nel grafo

A questo punto le chiavi del dizionario *relazioni* sono i vari identificatori presenti nel db e i valori le tabelle nelle quali compaiono.

Allo stesso tempo, le chiavi del dizionario *tabelle* sono i nomi delle tabelle e i valori tutti i campi presenti all'interno di esse.

Ora concentriamoci sulla ricerca dei campi presenti in più tabelle che non sono chiavi.

Questo passaggio è molto semplice, basta scorrere tutte le chiavi del dizionario *tabelle* e i suoi valori e appena si trova per la seconda volta lo stesso attributo (controllando ovviamente che non sia una chiave) lo si aggiunge alla lista dei ripetuti.

Successivamente,quest'ultimi devono essere aggiunti al dizionario *relazioni* poichè dovremo trovare i loro archi che saranno in uscita verso tutti i nodi in cui compariranno come proprietà dato che sono campi della corrispettiva tabella nel db relazionale.L'unica proprietà di questi nuovi nodi sarà "nomecampo\_id" seguendo la logica del resto del progetto.

Vediamo un pò di pseudocodice per fare chiarezza sul come facciamo sì che i nuovi nodi vengano correttamente inseriti nel dizionario *relazioni*.

```
for tabella in tabelle.keys():
    for campo in ripetuti:
        se campo in tabelle[tabella]:
            se campo not in relazioni:
                allora relazioni[campo]=[tabella]
            else:
                allora relazioni[campo].append(tabella)
```

**Figura 4.3.** Pseudocodice

Successivamente aggiungiamo ovviamente tutti nuovi nodi anche al dizionario *tabelle* con chiave il nome del campo e come valore la proprietà nomecampo\_id.

#### 4.1.3 Terzo step:Creazione del grafo

Andiamo quindi a creare un nuovo dizionario chiamato "grafo",dove effettivamente andremo a rappresentare il graph db,le quali chiavi sono i nomi dei nodi e i valori una lista di tuple contenenti il nome dell'arco e il nodo destinatario. Inoltre andiamo a creare anche una lista di nome "archi" che ci servirà per evitare di creare archi doppi(vogliamo evitare di avere due archi che collegano gli stessi nodi due volte) .

Adesso vediamo come mettere insieme tutte le informazioni acquisite fino ad ora tramite lo pseudocodice.

```

grafo={}

archi= []

for tabella in tabelle.keys():
    for campo in relazioni:
        se nometabella in relazioni[campo] and nometab!=campo:
            allora for tabella2 in relazioni[campo]:
                arco=tabella + "_" + tabella2
                doppia=tabella2 + "_" + tabella
                se arco not in archi and doppia not in archi and tabella!=tabella2:
                    allora grafo[tabella].add((tabella2,arco))
                    archi.append(arco)

```

**Figura 4.4.** Pseudocodice

#### 4.1.4 Quarto step:traduzione del grafo in QUERY Cypher

A questo punto non rimane che tradurre il grafo in una query Cypher, ricordando che abbiamo velocizzato il tutto traducendo nello step uno già tutte le istanze delle tabelle visitate scrivendole nella variabile *nodi*.

Mancano all'appello però ancora i nodi che non erano ancora presenti all'inizio, cioè quelli "nati" da attributi presenti in più di una tabella. Dopo aver effettuato questo controllo e aggiunto alla stringa *nodi* quest'ultimi possiamo passare alla traduzione delle relazioni di cui riporto il semplice pseudocodice anche per visualizzare così la sintassi della creazione degli archi in Cypher.

```

for nodo in grafo:

    for tupla in grafo[nodo]:

        rel+="MATCH(a:" + nodo + "),(b:" + tupla[0] + ") CREATE(a)-[r:" + tupla[1] + "]->(b);"
        rel+="\nWITH 1 as dummy \n"

filecypher=nodi+rel

```

**Figura 4.5.** Pseudocodice

Infine, non rimane che salvare in un file di estensione .cypher la stringa "filecypher" in modo da renderlo caricabile sul software NEO 4J. Per rendere il tutto più chiaro, riporto l'esempio della creazione di un nodo e di una relazione in linguaggio cypher.

```

nodo: CREATE(n:diagnoses_icd{subject_id:'11442057',hadm_id:'21518990',seq_num:'1',
icd_code:'65971',icd_version:'9'}).

```

```

relazione: MATCH(a:comments),(b:labevents) CREATE(a)-[r:comments_labevents]->(b);

```

## 4.2 Implementazione di predittore(path)

Iniziamo l'implementazione con la dichiarazione di tre variabili molto importanti ai fini dei nostri obiettivi:

- infop*: dizionario per mappare l'id dell'entrata in pronto soccorso(stay\_id) con i dati;
- terapiaintensiva*: insieme dove verranno inseriti gli hadm\_id delle persone entrate in terapia intensiva
- dizstay*: dizionario per mappare stay\_id con hadm\_id tra MIMIC IV e MIMIC ED

### 4.2.1 Raccolta dei dati dalle diverse tabelle

Come per lo script csvToNeo precedentemente descritto facciamo uso della funzione `os.walk()` della libreria `os` di Python per esplorare i file e le cartelle dei progetti cercando precisamente due tabelle: `triage` e `patients`.

Dalla tabella "triage" estraiamo inizialmente il campo `stay_id` che identifica l'entrata e il ricovero in pronto soccorso e la utilizziamo come chiave per il dizionario `infop`.

Utilizziamo il campo "stay\_id" e non il "subject\_id" poichè lo stesso soggetto potrebbe entrare diverse volte in pronto soccorso in più giorni differenti e a noi interessa dividere le due entrate trattandole separatamente in modo da non creare confusione con i dati di un paziente.

Proseguendo la raccolta dei dati dalla tabella "triage" andiamo ad inserire tutti i campi di nostro interesse all'interno dei valori di *infop*.

Proprio durante questo processo è stato fatto anche un lavoro di "sistemazione" di dati nel caso del campo "o2sat" (valore che come già descritto nel capitolo 3 il valore della saturazione dell'ossigeno) che sono stati raccolti in valori dal 90 al 100,assimilando tutti i valori <90 come uguali a 90 poichè sotto la soglia definita dai medici critica di respirazione dell'essere umano. Un altro motivo controllo aggiunto è su gli errori di data entry sempre nel seguente campo. Osservando che all'interno delle istanze ci sono valori >100 (il che logicamente è impossibile) ho deciso in fase di elaborazione dei dati di non prendere in considerazione tali istanze. Successivamente, dalla tabella "patients" estraiamo invece i dati anagrafici di nostro interesse,cioè "age" e "gender".

### 4.2.2 Mapping dei ricoveri e raccolta dati pazienti in terapia intensiva

Utilizziamo la tabella "edstays" per effettuare il mapping degli stay\_id con i relativi hadm\_id ed inserire la mappatura all'interno di `dizstay`(chiave=stay\_id,valore= hadm\_id corrispondente). Vediamo un pò di pseudocodice per rendere le cose più chiare.

```
se nometabella=="edstays":  
  
    allora apri il file csv e scorilo  
  
    for riga in file:  
  
        dizstay[stay_id]=riga[hadm_id]
```

**Figura 4.6.** Pseudocodice

A questo punto avremo nella variabile *dizstay* la mappatura degli *stay\_id* con i corrispettivi *hadm\_id* corretti in modo che i dati di entrambi i progetti corrispondano e che i ricoveri in pronto soccorso siano tracciabili anche in altri reparti, come nel nostro caso la terapia intensiva. A questo punto possiamo cercare la tabella "icustays" dove sono raccolti tutti i dati della terapia intensiva e raccogliere tutti gli "hadm\_id" presenti nell'insieme denominato "terapiaintensiva". Vediamo lo pseudocodice.

```
se nometabella=="icustays":

    allora apri il file csv e scorilo

    for riga in file:

        aggiungi alla lista terapiaintensiva l'hadm_id
```

**Figura 4.7.** Pseudocodice

### 4.2.3 Assegnazione valore al campo icu e creazione del file csv

Adesso non rimane che mettere insieme tutte le informazioni ottenute ed elaborarle in un file csv determinando il valore del campo icu per ogni istanza.

Come è facilmente intuibile la determinazione del valore è molto semplice, basta controllare, per ogni riga, se il proprio "hadm\_id" è presente nell'insieme *terapiaintensiva*. In caso di riscontro positivo allora il valore icu sarà uguale a 1, altrimenti 0.

Vediamo lo pseudocodice che riassume entrambe le cose in maniera chiara ed efficiente.

```
tabella=[]
header=["gender","age","temperature","heartrate","resprate","o2sat","sbp","dbp","pain","acuity","icu"]
for ricovero in infop:
    dati=[]
    aggiungi a dati infop[ricovero][0] (gender)
    aggiungi a dati infop[ricovero][1] (age)
    aggiungi a dati infop[ricovero][2] (temperature)
    .... aggiunta di tutti i campi restanti estrapolandoli da infop
    se dizstay[ricovero] in terapiaintensiva:
        allora aggiungi a dati il valore 1
    altrimenti:
        aggiungi a dati il valore 0
    aggiungi dati alla tabella
scrivi su file csv(header)
scrivi su file csv(tabella)
```

**Figura 4.8.** Pseudocodice

Ora non ci resta che eseguire lo script per avere in output il file result.csv correttamente creato

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	gender	age	temperature	heart rate	respiration	o2sat	sbp	dbp	pain	acuity	icu								
2	M	39	986.000	880.000	180.000	97.0	1.380.000	840.000	0.0000	30.000	0								
3	F	69	968.000	560.000	150.000	100.0	1.130.000	710.000	80.000	20.000	0								
4	F	56	990.000	550.000	180.000	100.0	2.280.000	1.020.000	100.000	20.000	0								
5	M	91	989.000	800.000	180.000	100.0	1.180.000	500.000	30.000	20.000	0								
6	F	91	971.000	620.000	150.000	98.0	1.160.000	450.000	0.0000	20.000	1								
7	F	56	980.000	680.000	180.000	100.0	1.320.000	670.000	100.000	30.000	0								
8	F	25	983.000	680.000	400.000	99.0	1.010.000	560.000	70.000	30.000	0								
9	F	28	978.000	1.140.000	180.000	99.0	1.360.000	740.000	100.000	20.000	0								
10	M	68	998.000	760.000	180.000	97.0	1.100.000	690.000	0.0000	30.000	0								
11	F	48	979.000	830.000	180.000	100.0	1.140.000	690.000	0.0000	20.000	0								
12	F	22	984.000	760.000	160.000	99.0	1.120.000	760.000	70.000	30.000	0								
13	M	66	984.000	840.000	180.000	95.0	1.220.000	720.000	80.000	30.000	0								
14	F	64	984.000	700.000	180.000	100.0	1.270.000	710.000	0.0000	10.000	1								
15	F	20	989.000	890.000	160.000	100.0	1.170.000	530.000	0.0000	30.000	0								
16	M	41	1.011.000	1.200.000	160.000	94.0	1.470.000	1.040.000	30.000	20.000	0								
17	M	78	986.000	670.000	160.000	100.0	1.400.000	710.000	30.000	20.000	0								
18	F	53	972.000	850.000	160.000	99.0	1.690.000	850.000	110.000	40.000	0								
19	F	79	975.000	650.000	200.000	94.0	1.680.000	620.000	0.0000	20.000	0								
20	M	19	978.000	740.000	160.000	95.0	1.040.000	590.000	100.000	40.000	0								
21	F	43	977.000	760.000	180.000	96.0	920.000	550.000	0.0000	30.000	0								
22	F	82	991.000	880.000	180.000	100.0	1.780.000	660.000	0.0000	20.000	1								
23	F	20	994.000	1.000.000	180.000	94.0	1.300.000	720.000	30.000	30.000	0								
24	F	20	975.000	600.000	160.000	97.0	900.000	600.000	130.000	30.000	0								
25	M	87	983.000	690.000	160.000	96.0	1.490.000	690.000	0.0000	30.000	0								
26	F	49	980.000	850.000	200.000	100.0	1.520.000	990.000	100.000	20.000	0								
27	F	50	973.000	710.000	200.000	100.0	1.190.000	750.000	60.000	30.000	0								
28	F	62	977.000	860.000	200.000	99.0	1.110.000	680.000	0.0000	20.000	0								

Figura 4.9. File result.csv

#### 4.2.4 Descrizione della tabella result.csv

Diamo uno sguardo ai campi presenti all'interno della tabella result.csv:

- gender: sesso;
- age: età;
- temperature: temperatura corporea;
- heartrate: battito cardiaco;
- resperate: frequenza cardiaca;
- o2sat: saturazione;
- sbp: pressione sanguigna sistolica;
- dbp: pressione sanguigna diastolica;
- pain: livello di dolore avvertito dal paziente;
- acuity: valore di gravità assegnato dall'operatore sanitario(1=MAX,5= MIN);
- icu: indica se il paziente ha avuto bisogno della terapia intensiva oppure no.





## Capitolo 5

# Risultati degli esperimenti della trasformazione in grafo

Una volta sviluppato lo script "csvToNeo" che traduce i progetti MIMIC IV e MIMIC IV-ED da file csv in una query in linguaggio Cypher andiamo ad eseguirla direttamente nel programma Neo 4J così da poter avere una visualizzazione completa della struttura dei progetti uniti.

### 5.1 Obiettivi

Gli obiettivi di questo esperimento sono diversi:

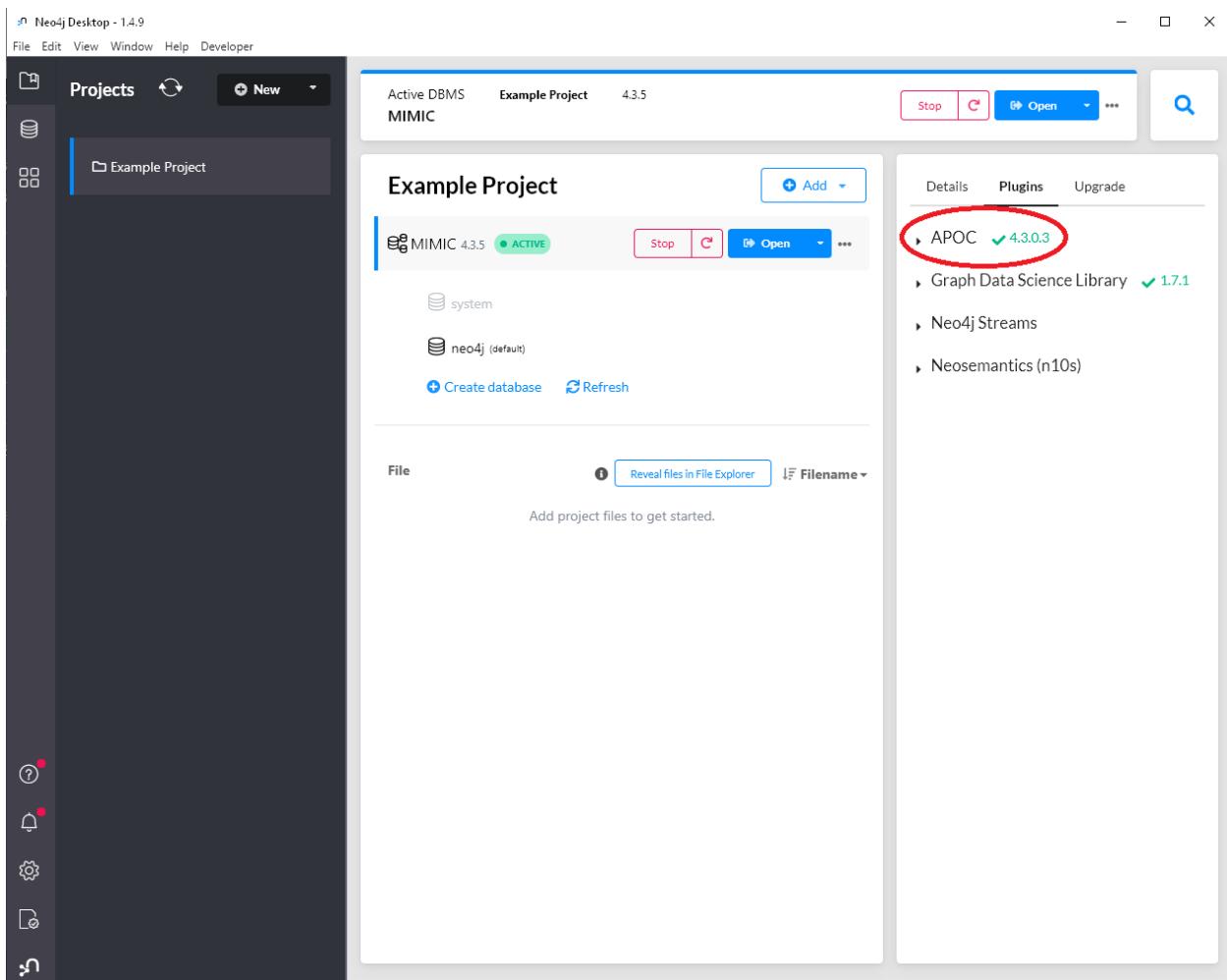
- il primo* è quello di riuscire ad ottenere la struttura dei progetti potendo avere un riscontro visivo(per poi successivamente valutare i campi appropriati da utilizzare nel predittore).
- Il secondo* è quello di unire i due progetti e poter controllare che la loro unione sia omogenea attraverso le query.
- Il terzo* è ottenere statistiche interessanti sui nodi e sulle relazioni.

## 5.2 Caricamento e impostazione della query in NEO 4J

Per eseguire il caricamento della query Cypher nel programma NEO 4J abbiamo bisogno precedentemente di effettuare diversi passaggi preparativi.

Innanzitutto dobbiamo installare il plugin APOC all'interno del nostro progetto.

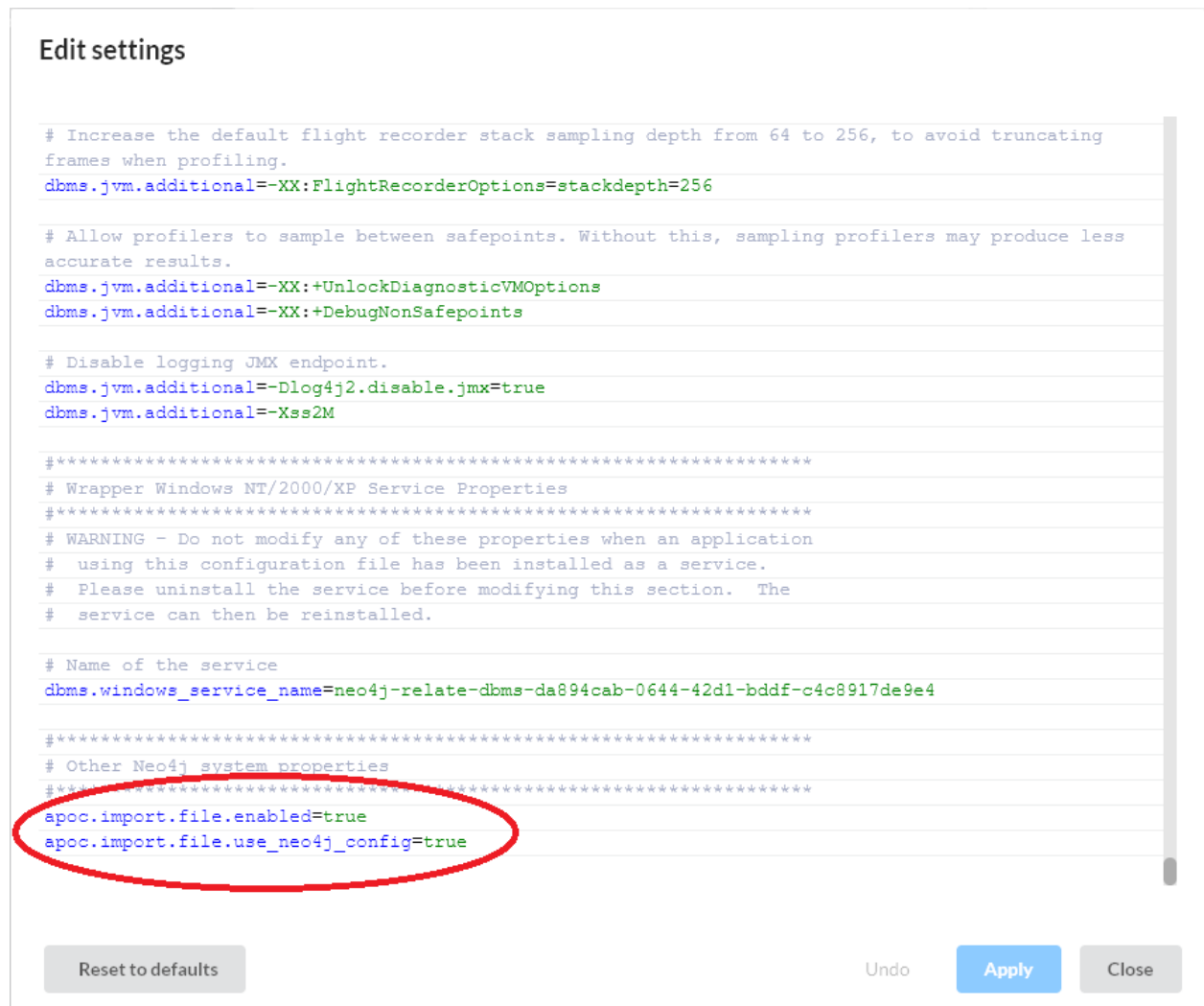
La libreria APOC è composta da molte (circa 450) procedure e funzioni per aiutare con molte attività diverse in aree come l'integrazione dei dati, gli algoritmi grafici o la conversione dei dati. Una volta installato correttamente apparirà come nell'immagine seguente:



**Figura 5.1.** Plugin APOC installato correttamente

A questo punto, andiamo ad aggiungere le seguenti righe all'interno dei "settings" del nostro progetto:

```
-apoc.import.file.enabled=true
-apoc.import.file.use_neo4j_config=true
```



**Figura 5.2.** Linee correttamente inserite all'interno dei settings del nostro progetto

Ora non ci resta che caricare il file Cypher(grafo.cypher) prodotto dal nostro script "csvToNeo" avviando il Neo4J Browser e digitando nella linea di comando:

```
-CALL apoc.cypher.runFile("grafo.cypher");
```

Una volta eseguito il comando i nodi e le relazioni saranno correttamente create.

## 5.3 Correttezza

Per verificare la correttezza del grafo dobbiamo controllare principalmente che:

- Il grafo finale rispecchi la struttura dell'organizzazione dei database descritta nei progetti.
- Gli attributi con lo stesso nome che sono ripetuti in più di una tabella devono diventare dei nodi(a meno che non siano attributi chiave come già visto nel Capitolo 4).
- Che i due progetti siano effettivamente collegati nel grafo e che siano quindi raggiungibili da una tabella di MIMIC IV ad una di MIMIC-ED.

### 5.3.1 Verifica struttura MIMIC IV

Partiamo col verificare la struttura del progetto MIMIC IV [\[1\]](#).

Utilizziamo nel NEO 4J browser il seguente comando per visualizzare tutte le label dei nodi presenti nel nostro grafo:

-CALL db.labels()

Ora confrontiamo la strutture e le tabelle descritte nella descrizione del progetto ufficiale con quelle del nostro grafo.

Essenzialmente il progetto MIMIC IV è diviso in 3 sottocartelle dal nome(come visto in maniera più approfondita nel Capitolo 2 "Background"):

-*Core*

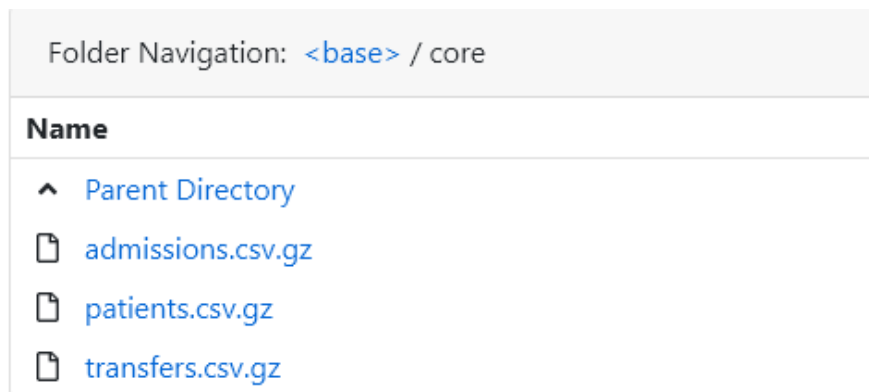
-*Hosp*

-*Icu*

Facciamo quindi ora le nostre verifiche confrontando i risultati prodotti dal Neo 4J Browser con le descrizioni ufficiali delle tre cartelle e delle tabelle che contengono per controllare che siano tutte presenti nel grafo.

### 5.3.2 Verifica di "Core"

Procediamo adesso con la verifica della cartella *Core*, che ricordiamo essere il "cuore" del progetto, controllando se tutte le sue tabelle sono effettivamente presenti all'interno del nostro Graph DB.



(a) Tabelle della cartella "Core"



(b) Label appartenenti alla cartella "Core" presenti all'interno del grafo

**Figura 5.3.** Come possiamo vedere la struttura della cartella Core è rappresentata all'interno del grafo

### 5.3.3 Verifica di "Hosp"

Procediamo adesso con la verifica della cartella *Hosp* controllando se tutte le sue tabelle sono effettivamente presenti all'interno del nostro Graph DB.

Folder Navigation: <base> / hosp	
Name	
^	Parent Directory
📄	d_hcpcs.csv.gz
📄	d_icd_diagnoses.csv.gz
📄	d_icd_procedures.csv.gz
📄	d_labitems.csv.gz
📄	diagnoses_icd.csv.gz
📄	drgcodes.csv.gz
📄	emar.csv.gz
📄	emar_detail.csv.gz
📄	hcpcsevents.csv.gz
📄	labevents.csv.gz
📄	microbiologyevents.csv.gz
📄	pharmacy.csv.gz
📄	poe.csv.gz
📄	poe_detail.csv.gz
📄	prescriptions.csv.gz
📄	procedures_icd.csv.gz
📄	services.csv.gz

**Figura 5.4.** Tabelle della cartella "Hosp"

neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

File Edit View Window Help Developer

```
neo4j$ CALL db.labels()
```

	label
4	"diagnoses_icd"
5	"drgcodes"
6	"d_hcpcs"
7	"d_icd_diagnoses"
8	"d_icd_procedures"
9	"d_labitems"
10	"emar"
11	"emar_detail"
12	"hpcsevents"
13	"labevents"
14	"microbiologyevents"
15	"pharmacy"

**Figura 5.5.** Prima parte delle label appartenenti alla cartella "Hosp" presenti all'interno del grafo

neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

File Edit View Window Help Developer

```
neo4j$ CALL db.labels()
```

	label
16	"poe"
17	"poe_detail"
18	"prescriptions"
19	"procedures_icd"
20	"services"

**Figura 5.6.** Seconda parte delle label appartenenti alla cartella "Hosp" presenti all'interno del grafo

Come possiamo osservare dal confronto dei file csv presenti nel progetto e le label presenti nel grafo, la cartella *Hosp* è correttamente rappresentata all'interno del nostro Graph DB.



### 5.3.4 Verifica di "Icu"

Procediamo adesso con la verifica della cartella *Icu* controllando se tutte le sue tabelle sono effettivamente presenti all'interno del nostro Graph DB.

Folder Navigation: <base> / icu	
Name	
^	Parent Directory
📄	chartevents.csv.gz
📄	d_items.csv.gz
📄	datetimeevents.csv.gz
📄	icustays.csv.gz
📄	inputevents.csv.gz
📄	outputevents.csv.gz
📄	procedureevents.csv.gz

(a) Tabelle della cartella "Icu"

neo4j@bolt://localhost:7687/neo4j - Neo4j Browser	
File Edit View Window Help Developer	
neo4j\$ CALL db.labels()	
label	
21	"chartevents"
22	"datetimeevents"
23	"d_items"
24	"icustays"
25	"inputevents"
26	"outputevents"
27	"procedureevents"

(b) Label appartenenti alla cartella "Icu" presenti all'interno del grafo

**Figura 5.7.** Come possiamo vedere la struttura della cartella Icu è rappresentata all'interno del grafo

### 5.3.5 Verifica struttura di MIMIC-ED

Questo progetto<sup>[2]</sup> non è suddiviso in sottocartelle perchè molto piccolo rispetto al precedente (funziona per l'appunto da integrazione per MIMIC IV) e la verifica è immediata come possiamo facilmente osservare.

Folder Navigation: <base> / ed	
Name	
^	Parent Directory
📄	diagnosis.csv.gz
📄	edstays.csv.gz
📄	medrecon.csv.gz
📄	pyxis.csv.gz
📄	triage.csv.gz
📄	vitalsign.csv.gz

(a) Descrizione di MIMIC-ED

neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

File Edit View Window Help Developer

```
neo4j$ CALL db.labels()
```

	label
28	"diagnosis"
29	"edstays"
30	"medrecon"
31	"pyxis"
32	"triage"
33	"vitalsign"

(b) Label appartenenti alla cartella "Icu" presenti all'interno del grafo

**Figura 5.7.** Come possiamo osservare la struttura del progetto MIMIC-ED è rappresentata con successo.

### 5.3.6 Verifica sui campi ripetuti

Tutto il resto delle label sono i campi presenti in piu' di una tabella del db che per il metodo di trasformazione utilizzato(vedi capitolo 3 Metodi ) sono diventati tali.

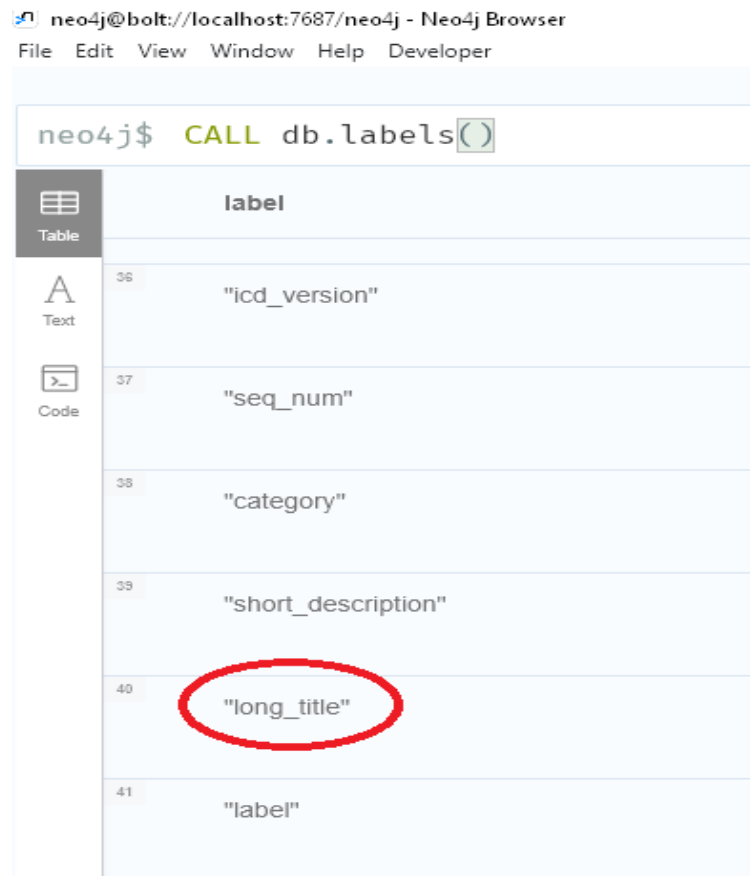
Di seguito ne riporto un esempio con il campo "long\_title".

	A	B	C	D	E	F	G
1	icd_code	icd_version	long_title				
2	10	9	Cholera due to vibrio cholerae				
3	11	9	Cholera due to vibrio cholerae el tor				
4	19	9	Cholera, unspecified				
5	20	9	Typhoid fever				
6	21	9	Paratyphoid fever A				

(a) Campi tabella "d\_icd\_diagnoses"

	A	B	C
1	icd_code	icd_version	long_title
2	1	9	Therapeutic ultrasound of vessels of head and neck
3	2	9	Therapeutic ultrasound of heart
4	3	9	Therapeutic ultrasound of peripheral vascular vessels
5	9	9	Other therapeutic ultrasound
6	1	10	Central Nervous System and Cranial Nerves, Bypass

(b) Campi tabella "d\_icd\_procedures"



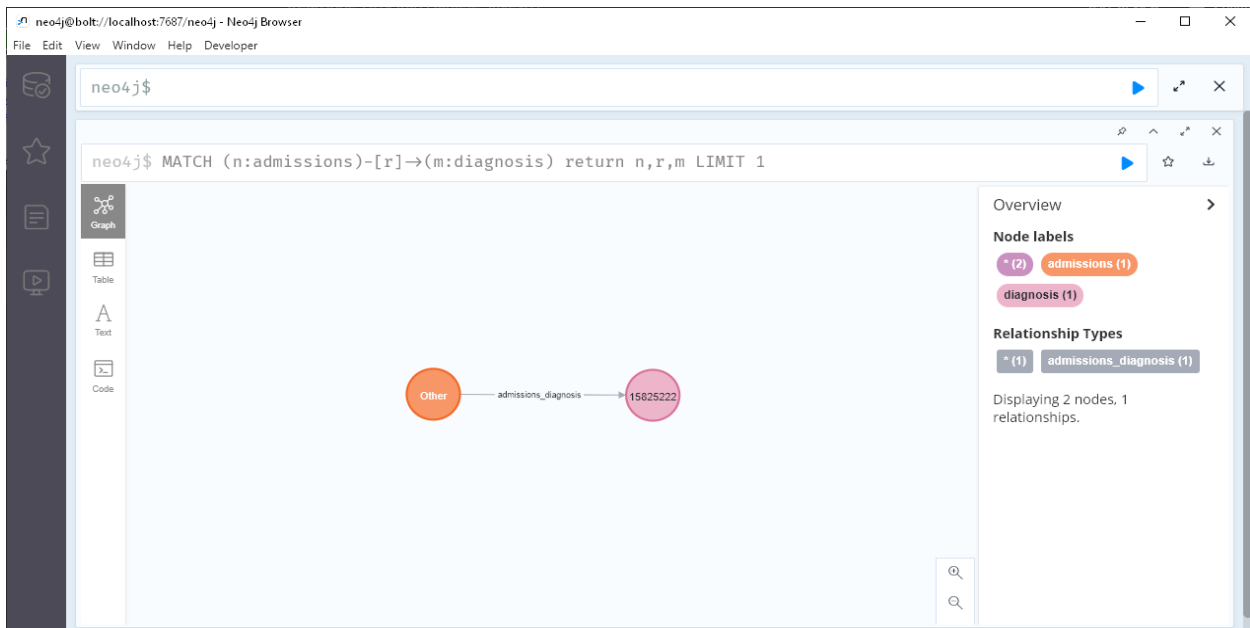
(c) "long\_title" è presente nelle label del nostro grafo

**Figura 5.7.** Essendo presente in più tabelle il campo "long\_title" diventa un nodo del grafo

### 5.3.7 Verifica collegamento tra i MIMIC IV e MIMIC-ED

Effettuiamo ora una verifica del collegamento tra i due progetti MIMIC-IV e MIMIC-ED. Inizialmente osserviamo come i nodi con label "admissions"(MIMIC IV) e "diagnosis"(MIMIC-ED) siano collegati tra loro tramite la seguente QUERY d'esempio:

```
MATCH (n:admissions)-[r]->(m:diagnosis) return n,r,m LIMIT 1
```



**Figura 5.8.** Esecuzioni della QUERY in Neo 4J

Adesso verifichiamo come effettivamente dai nodi di tipo "diagnosis" si possano raggiungere tutti gli altri nodi appartenenti al progetto MIMIC-ED. Effettueremo la verifica eseguendo la seguente QUERY:

```
MATCH (n:diagnosis)-[r]->(m) return distinct labels(m)
```



**Figura 5.9.** Esecuzioni della QUERY in Neo 4J



**Figura 5.10.** Esecuzioni della QUERY in Neo 4J dal punto di vista grafico

Come possiamo osservare dall'immagine le label ritornate sono 6 che sono proprio quelle del progetto MIMIC-ED.

## 5.4 Valutazioni Tecniche

A questo punto non ci rimane che fare qualche valutazione tecnica interessante sul grafo ottenuto. Con l'aiuto del software NEO 4J e delle QUERY in linguaggio Cypher possiamo estrarre diverse informazioni dal nostro grafo(ricordando quanto detto nel capitolo 4 "Implementazione del limite impostato alle righe).

### 5.4.1 Numero totale dei nodi

Per prima cosa vediamo il numero totale dei nodi nel nostro grafo con la seguente QUERY:

```
MATCH (n) RETURN count(n)
```



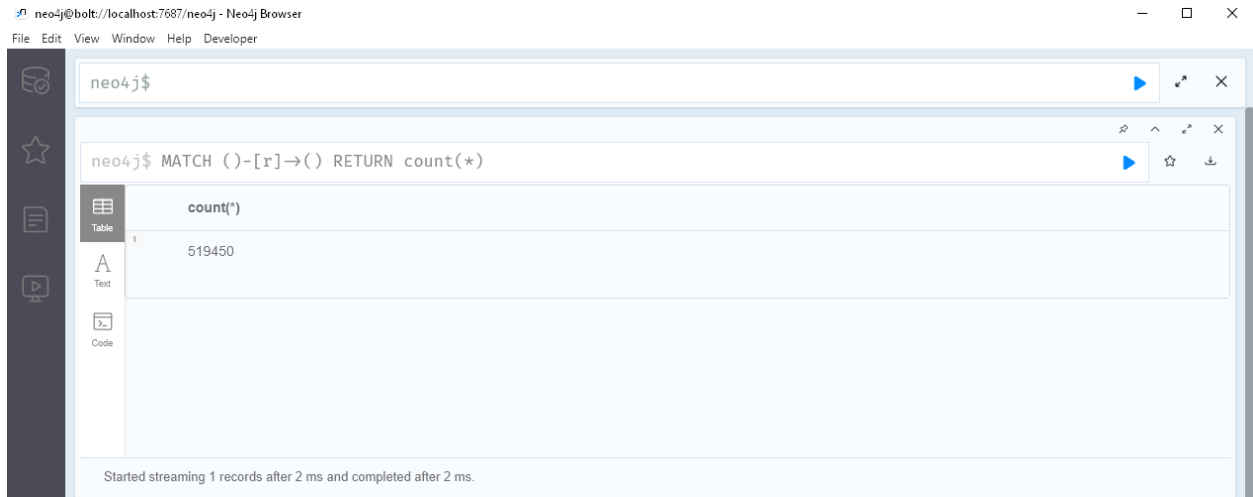
**Figura 5.11.** QUERY che ritorna il numero di nodi totali del grafo



### 5.4.2 Numero totale delle relazioni

Ora vediamo la QUERY per contare il numero di relazioni presente all'interno del nostro grafo:

```
MATCH ()-[r]->() RETURN count(*)
```

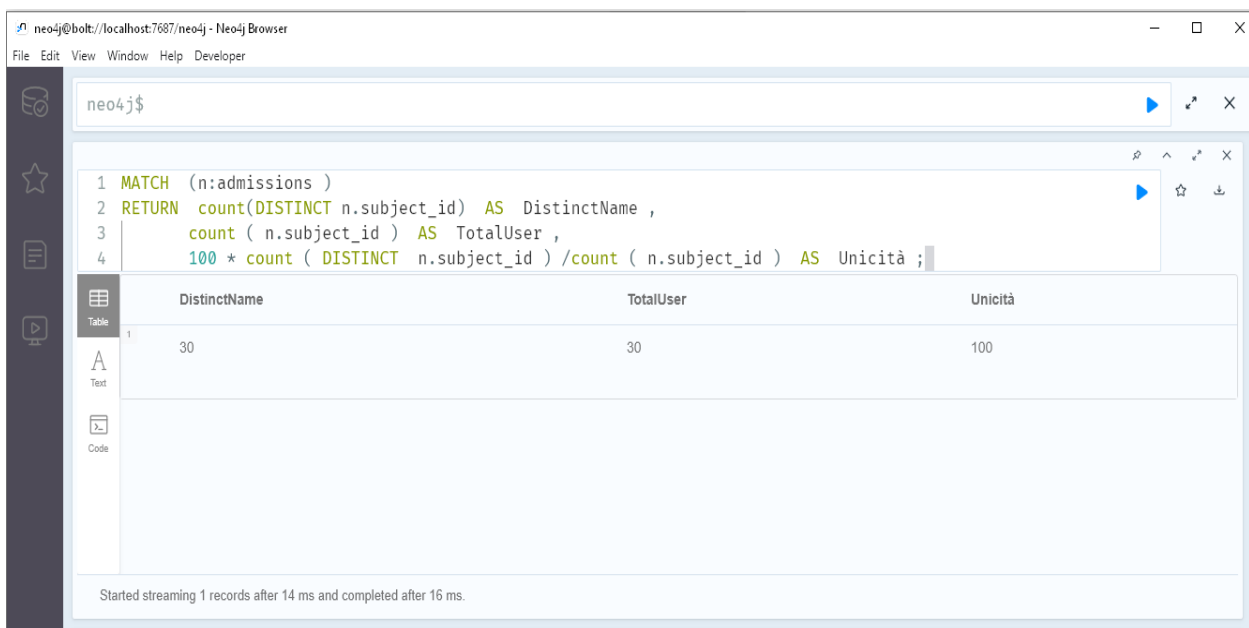


**Figura 5.12.** QUERY che ritorna il numero di relazioni totali presenti nel grafo

### 5.4.3 Unicità delle proprietà

La seguente QUERY può essere molto utile poichè potrebbe essere utilizzata per valutare i candidati ad essere una chiave ID:

```
MATCH (n:admissions)
RETURN count(DISTINCT n.subject_id) AS DistinctName,
count (n.subject_id) AS TotalUser,
100 * count(DISTINCT n.subject_id)/count (n.subject_id) AS Unicità ;
```



The screenshot shows the Neo4j Browser interface. The top bar indicates the URL 'neo4j@bolt://localhost:7687/neo4j - Neo4j Browser'. The left sidebar contains icons for home, star, list, play, and code. The main area displays a Cypher query in a text editor, which has been executed. Below the query, the results are shown in a table view. The table has three columns: 'DistinctName', 'TotalUser', and 'Unicità'. There is one data row with values 30, 30, and 100 respectively. At the bottom, a status message reads: 'Started streaming 1 records after 14 ms and completed after 16 ms.'

	DistinctName	TotalUser	Unicità
1	30	30	100

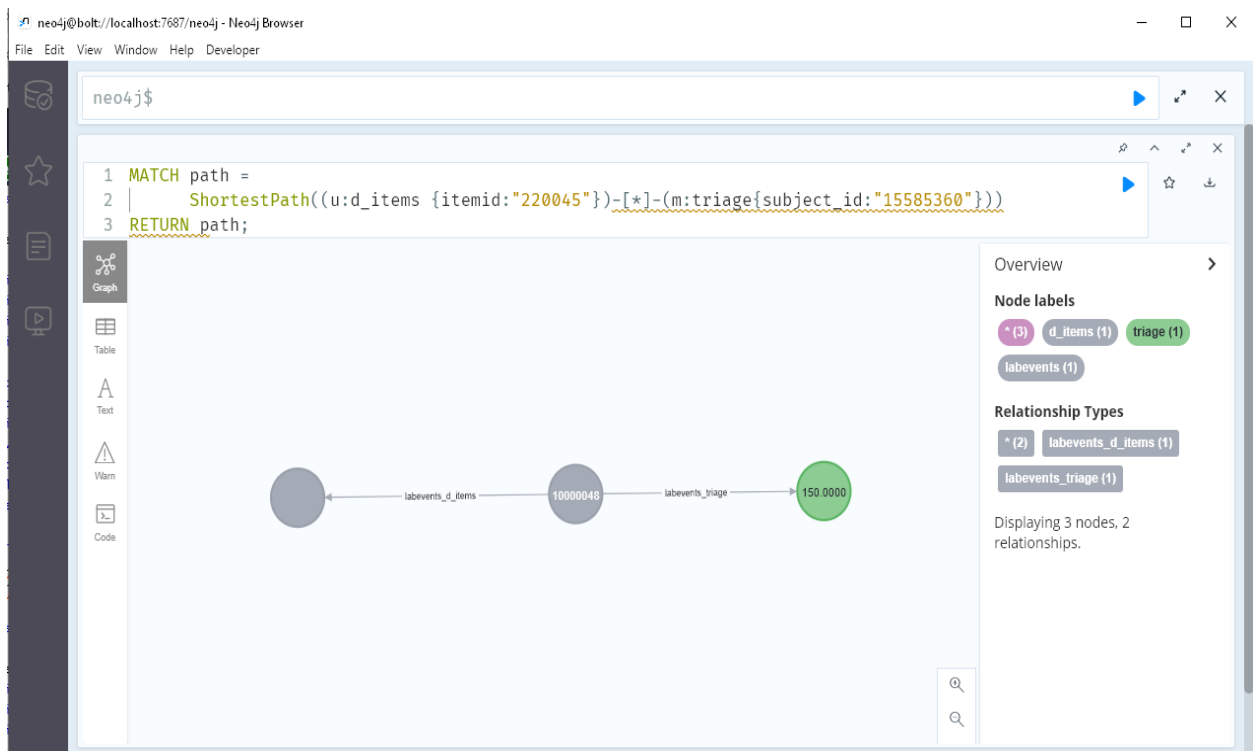
**Figura 5.13.** Risultati della QUERY che calcola l'unicità delle istanze di una proprietà

In questo caso prendiamo come tutti i nodi con label "admissions" e valutiamo l'unicità della proprietà "subject\_id". Quest'ultima risulta un'ottima candidata per essere l'ID poichè, come possiamo vedere dai risultati della QUERY, abbiamo un 100% di unicità (in pratica non ci sono mai due valori ripetuti tra tutte le 30 istanze).

#### 5.4.4 Il percorso più corto tra due nodi

La seguente QUERY ci aiuta a trovare il percorso più corto tra due nodi:

```
MATCH path =  
ShortestPath((u:d_items{itemid:"220045"})-[*]-(m:triage{subject_id:"15585360"}))  
Return path;
```



**Figura 5.14.** Risultati della QUERY che calcola il percorso più breve da un nodo all'altro.

Il percorso più breve tra due nodi ci dice che i due nodi sono collegati. Questo non è necessariamente l'unico percorso ma con la seguente QUERY abbiamo come risultato il più corto in assoluto. Questa QUERY è molto potente poichè in linea generale può essere molto utile per valutare la struttura e l'ottimizzazione del grafo.



## Capitolo 6

# Risultati degli esperimenti sul predittore

Dopo aver implementato lo script per raccogliere tutti i dati vitali del paziente in entrata al pronto soccorso e successivamente aver assegnato con criterio i giusti valori al campo "icu"(0 o 1), con l'aiuto del software "Weka" ho testato il nostro file "result.csv" con un algoritmo di classificazione binaria J48 per osservare la correttezza dei risultati prodotti dall'algoritmo e ottenere l'albero decisionale.

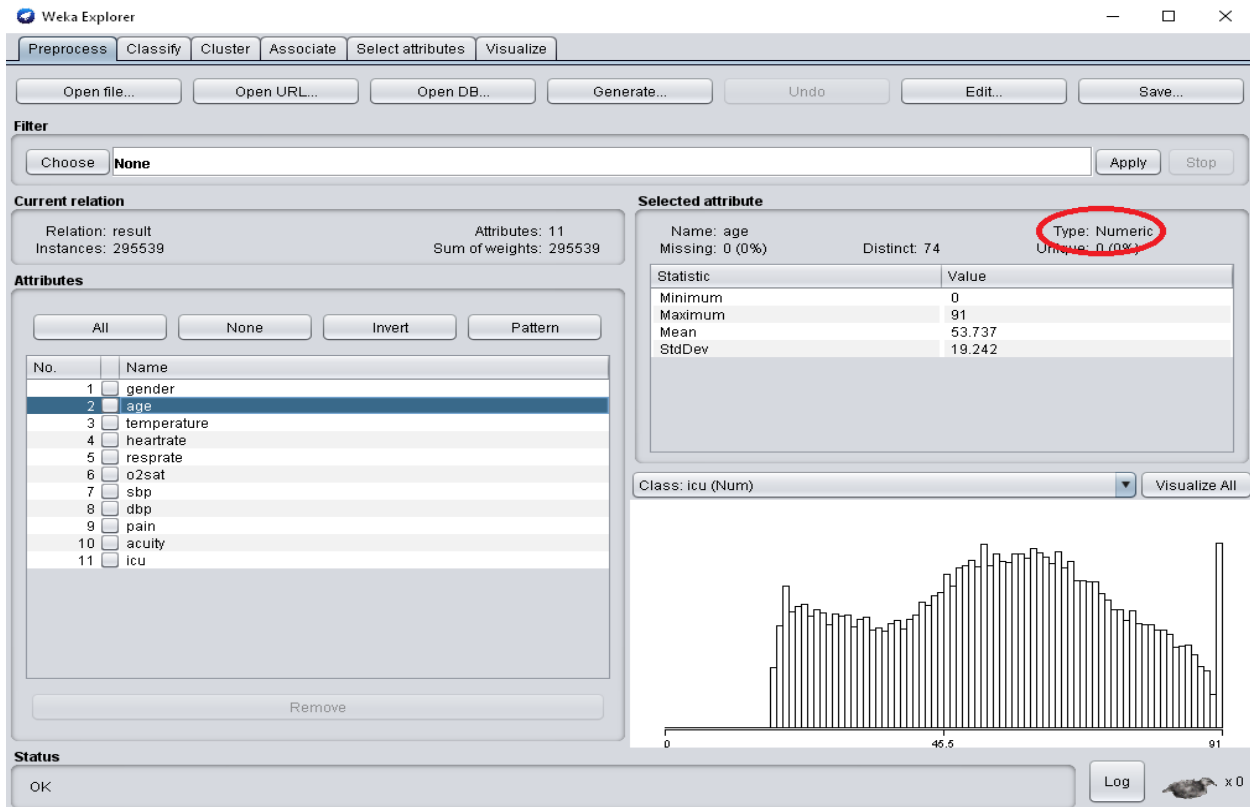
### 6.1 Obiettivi

Una volta capita la struttura dei progetti e come interpretare i dati che ci sono al suo interno l'obiettivo principale del nostro esperimento diventa quello di ottenere, una volta eseguito l'algoritmo di classificazione J48 (semplice implementazione Java Open Source del famoso C4.5), non solo avere una predizione con il tasso di errore più basso possibile ma avere una predizione coerente in entrambi i casi, che siano 0 (paziente non entra in terapia intensiva) o che siano 1 (paziente entra in terapia intensiva).

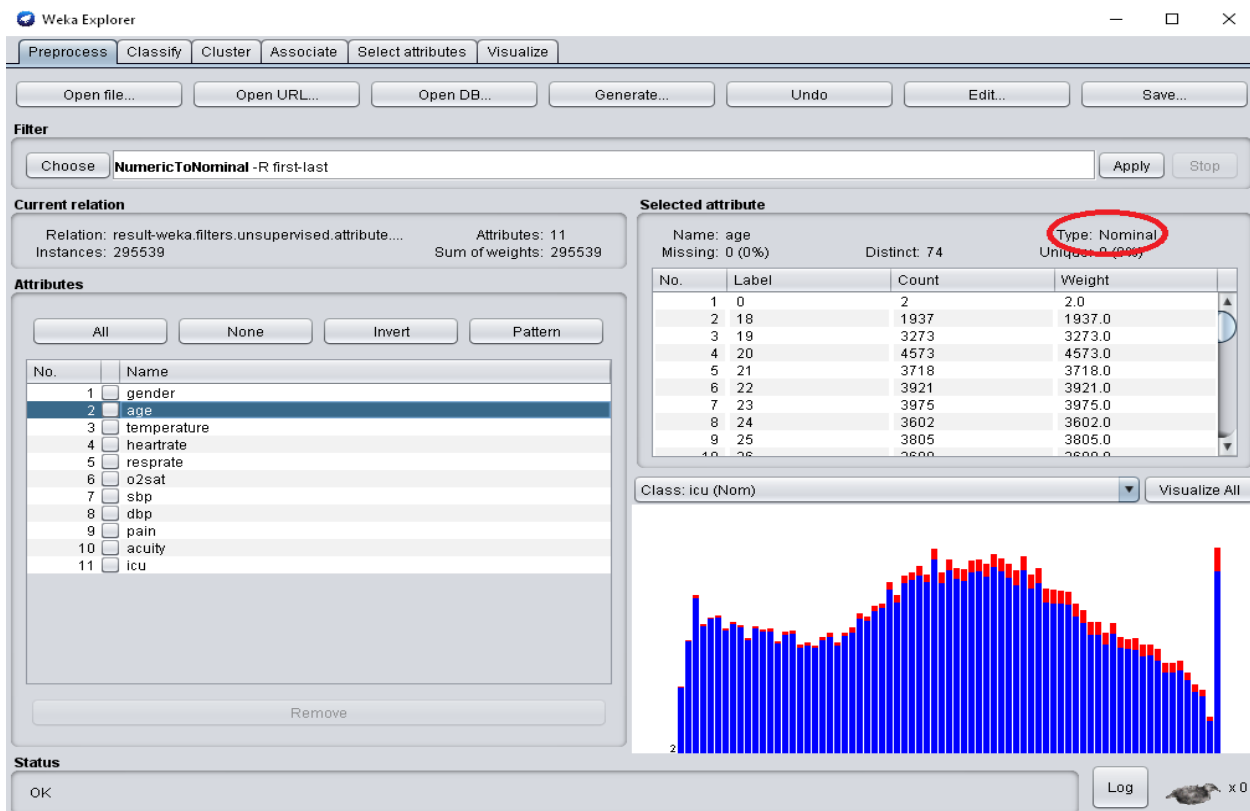
### 6.2 Settaggi

Trovare i giusti settaggi per rendere affidabile il risultato dopo l'esecuzione dell'algoritmo J48 è stato un processo che ha richiesto diverso tempo e diversi test. Partiamo dalla preparazione dei dati del file "result.csv" dove tutti gli attributi numerici sono stati "trasformati" in nominali dall'apposita sezione "Preprocess" di Weka grazie per l'appunto al filtro "NumericToNominal". Ciò è stato fatto per rendere possibile l'uso dell'algoritmo J48 che accetta solamente campi di tipo nominale.

### 6.2.1 Primo passaggio: conversione dei campi da numerici a nominali



(a) L'attributo "age" è numerico prima dell'esecuzione del filtro

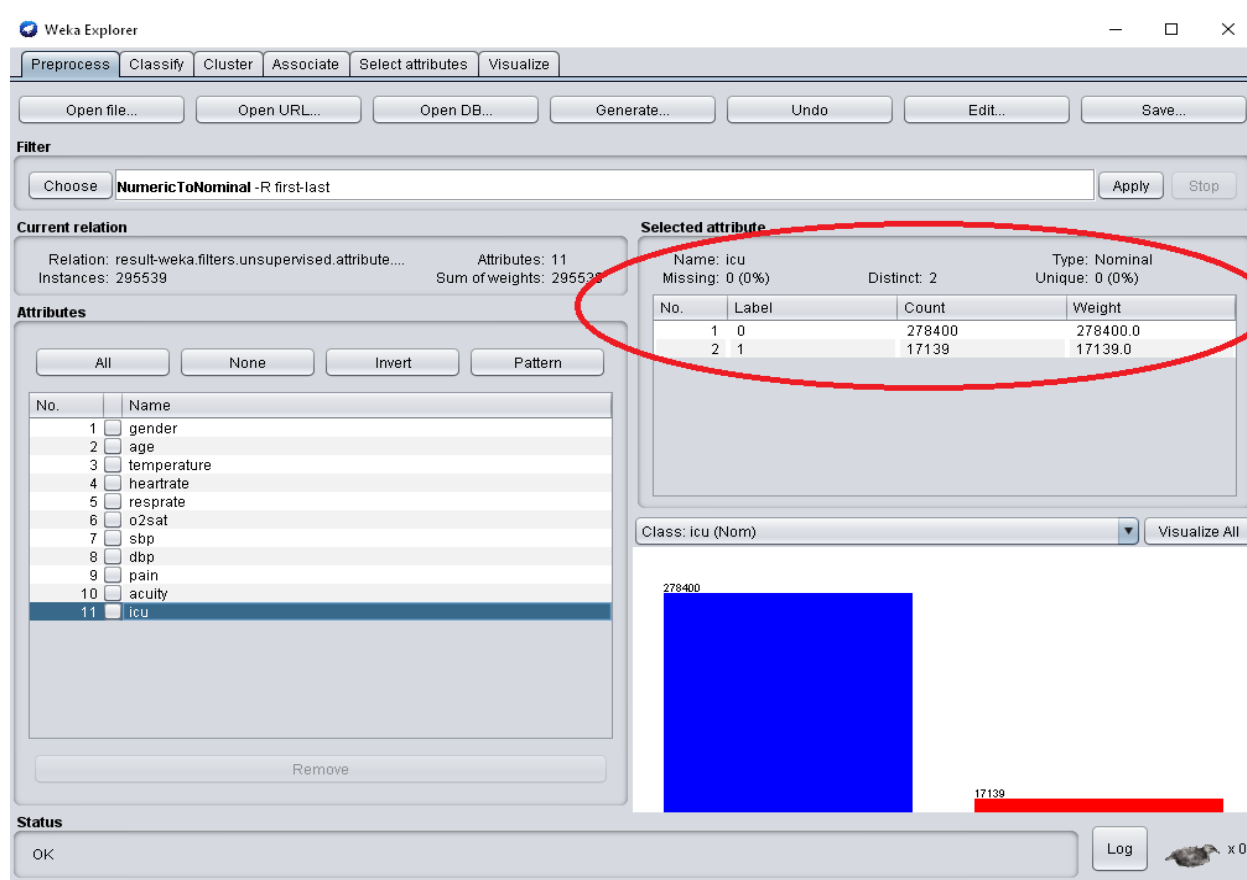


(b) L'attributo "age" diventa di tipo nominale dopo l'esecuzione del filtro

**Figura 6.1.** Primo step. Conversione dei campi da numerici a nominali

### 6.2.2 Secondo passaggio: bilanciamento delle istanze del campo "icu"

Il secondo passaggio, importantissimo per non falsare i risultati dell'albero decisionale, è stato caratterizzato dall'utilizzo del filtro "ClassBalancer" per bilanciare l'altezza di tutti i valori degli attributi. C'è stato bisogno di utilizzare tale filtro perchè i valori nelle istanze del nostro campo "icu" erano molto squilibrati (in favore ovviamente del valore 0). Tutto ciò portava ad avere un albero decisionale con percentuali altissime di correttezza della predizione (oltre il 93%) ma che sbagliava sempre nel caso nel quale il valore del campo "icu" fosse uguale a 1. Ciò falsificava i risultati poichè in ogni caso l'algoritmo assegnava valore 0 alla predizione mantenendo la percentuale di errore bassissima nonostante il tutto fosse completamente senza logica.



Weka Explorer

Preprocess | Classify | Cluster | Associate | Select attributes | Visualize

Open file... | Open URL... | Open DB... | Generate... | Undo | Edit... | Save...

Filter: Choose **NumericToNominal -R first-last** [Apply] [Stop]

Current relation: Relation: result-weka.filters.unsupervised.attribute... Attributes: 11 Instances: 295539 Sum of weights: 295539

Attributes: All | None | Invert | Pattern

No.	Name
1	gender
2	age
3	temperature
4	heartrate
5	resprate
6	o2sat
7	stp
8	dbp
9	pain
10	acuity
11	icu

Remove

Selected attribute: Name: icu Missing: 0 (0%) Distinct: 2 Type: Nominal Unique: 0 (0%)

No.	Label	Count	Weight
1	0	278400	278400.0
2	1	17139	17139.0

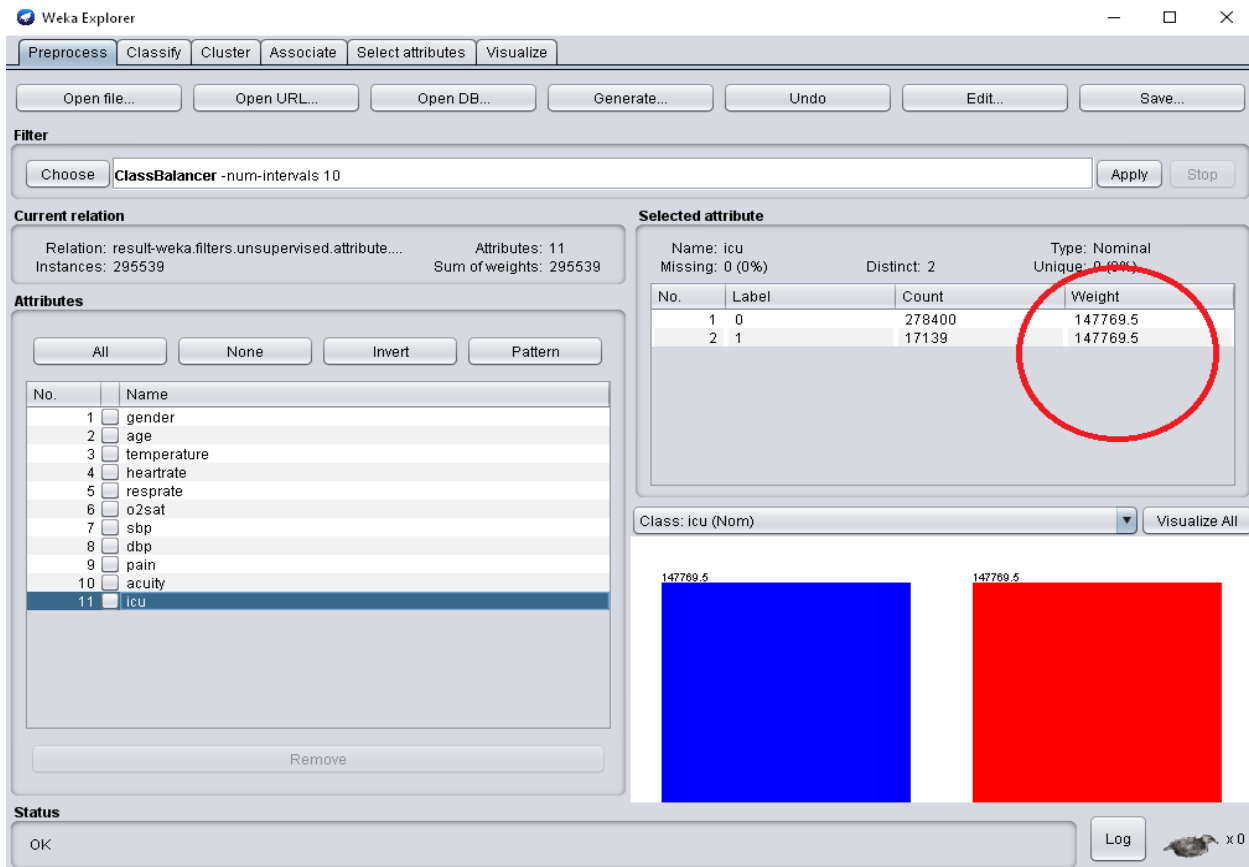
Class: icu (Nom) [Visualize All]

278400

17139

Status: OK [Log] x 0

(a) Notare la differenza di istanze di 0 e 1 (colonna "Count")



(b) Dopo l'esecuzione del filtro la colonna "Weight" ha stesso valore in entrambe le istanze

**Figura 6.1.** Secondo passaggio. "Bilanciamento" delle istanze del campo "icu"

### 6.2.3 Il classificatore J48

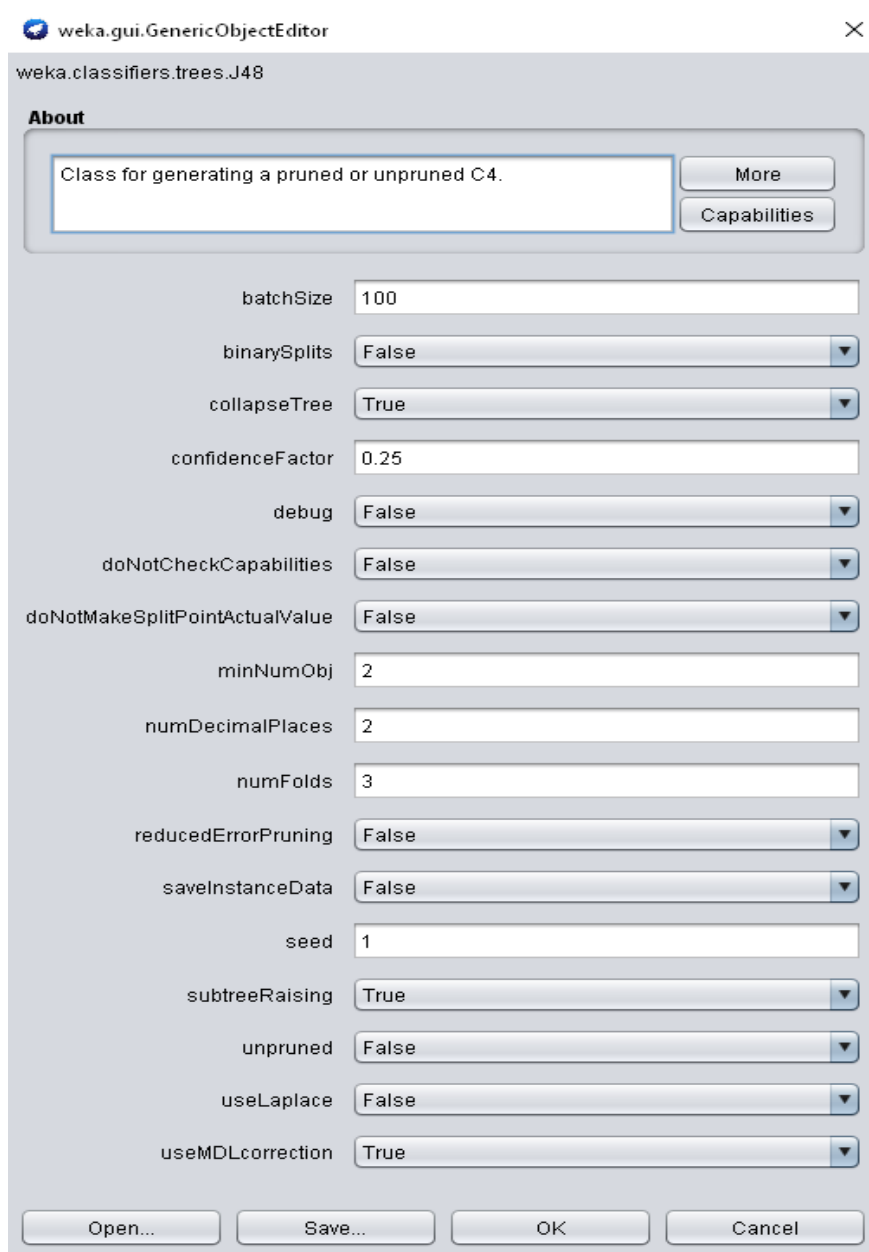
L'algoritmo [9](#) alla base del classificatore J48 è C4.5 il quale è un algoritmo di classificazione che produce alberi decisionali basati sulla teoria dell'informazione.

L'implementazione J48 dell'algoritmo C4.5 ha molte funzionalità tra cui la contabilizzazione dei valori mancanti, la potatura degli alberi decisionali (pruning), gli intervalli di valori degli attributi, la derivazione di regole, ecc. Nello strumento di data mining WEKA, J48 è un'implementazione Java open source di l'algoritmo C4.5. J48 consente la classificazione tramite alberi decisionali o regole generate da essi. Questo algoritmo costruisce alberi decisionali basati su un insieme di dati di training. I dati di training sono un insieme  $S = s_1, s_2, \dots$  di campioni già classificati. Ogni campione  $s_i$  è costituito da un vettore  $p$ -dimensionale  $(x_1, i, x_2, i, \dots, x_p, i)$  dove  $x_j$  rappresenta i valori degli attributi o le caratteristiche del campione corrispondente, nonché la classe in cui rientra il campione. Per ottenere la massima precisione di classificazione, l'attributo migliore su cui dividere è l'attributo con le informazioni maggiori. Ad ogni nodo dell'albero, l'algoritmo C4.5 sceglie l'attributo dei dati che più efficacemente suddivide il suo insieme di campioni in sottoinsiemi, posizionati in una classe o nell'altra. Per prendere la decisione viene scelto l'attributo con il guadagno di informazioni normalizzato più elevato. L'algoritmo C4.5 ricorre quindi ai sottoelenchi partizionati utilizzando un approccio divide et impera e crea un albero decisionale basato sull'algoritmo greedy.



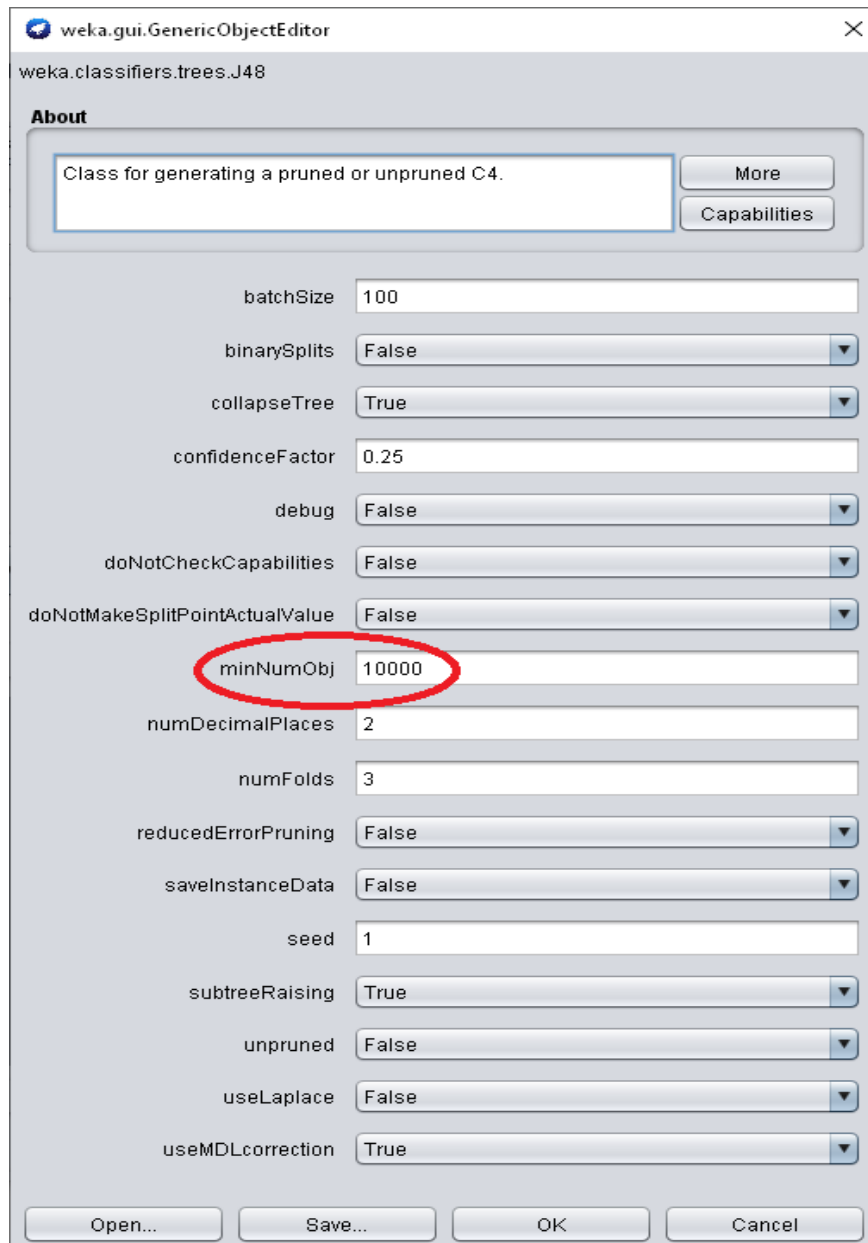
### 6.2.4 Settaggi pre-esecuzione J48

Ora passiamo ai settaggi veri e propri dell'algoritmo di classificazione binaria J48. Una volta entrati nella sezione "Classify" scegliamo il nostro algoritmo di classificazione, che nel nostro caso si trova nella sezione dei Decision Tree. Una volta selezionato ci troviamo davanti le seguenti impostazioni di default.



**Figura 6.2.** Settaggi iniziali algoritmo di classificazione J48

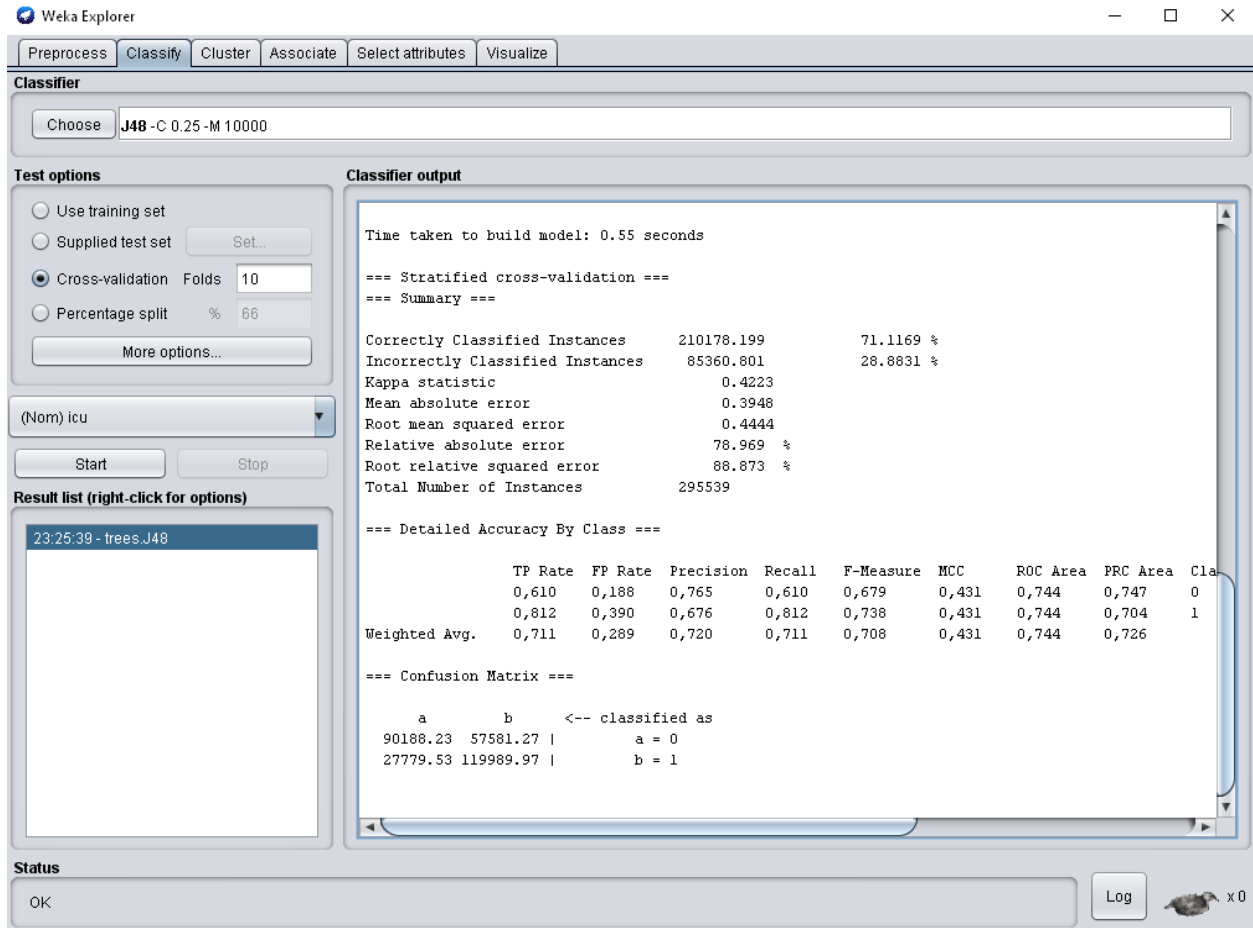
Di questi settaggi, dopo diverse prove, l'unica variabile che ho deciso di cambiare è la variabile "minNumObj" che da valore=2 passa ad avere valore=10.000. Questo è un'impostazione importantissima per rendere l'albero decisionale più piccolo possibile e a sua volta più leggibile poichè impone che in ogni foglia debbano arrivare come minimo n(valore assegnato alla variabile) istanze. Un altro campo molto importante, ma che dopo i test ho deciso di lasciare così, è "unpruned" lasciato con valore assegnato False. Questa variabile se impostata a True fa sì che l'algoritmo non semplifichi l'albero e crei quanti più nodi possibili rendendolo folto di foglie e meno leggibile, potrebbe essere un buon compromesso nel caso in cui la sua attivazione porti ad un particolare miglioramento della correttezza a discapito delle prestazioni, ma nel nostro caso essendo il miglioramento misero ho optato per lasciarlo a False.



**Figura 6.3.** Settaggi finali, l'algoritmo è pronto per essere lanciato

## 6.3 Valutazione sulla correttezza del predittore

Per quanto riguarda la correttezza dell'algoritmo decisionale i risultati ottenuti sono i seguenti.



**Figura 6.4.** Risultati ottenuti dopo l'esecuzione di J48

I due valori che maggiormente ci interessano sono i seguenti:

- Istanze classificate correttamente: 71.1169 %
- Istanze classificate incorrettamente: 28.8831 %

Questi due valori ci indicano come il nostro albero decisionale prodotto sarà piuttosto affidabile e produca risultati veritieri. Altri valori interessanti sono quelli di precisione relativi alle due classi 0 e 1.

- Precisione classe "0":0.765
- Precisione classe "1":0.676
- Media della precisione:0.720

Questi due dati sono quelli che,insieme ai due precedenti, sono gli indicatori della bontà dei risultati ottenuti poichè ci dimostrano che gli errori commessi sono bilanciati su entrambi i valori possibili di "icu",questo non succedeva prima di usufruire del filtro "Class Balancer" quando la percentuale di istanze classificate correttamente era vicina pressochè al 93 % ma praticamente l'algoritmo restituiva sempre "0" come risultato non prendendo proprio in considerazione la possibilità di prevedere il valore "1" creando di fatto un semplice albero decisionale con un solo nodo e una sola foglia dal valore "0" come testimoniano i seguenti dati prodotti dall'esecuzione del J48 senza l'uso del filtro.

Weka Explorer

Preprocess Classify Cluster Associate Select attributes Visualize

**Classifier**

Choose **J48 -C 0.25 -M 10000**

**Test options**

☐ Use training set  
☐ Supplied test set Set...  
☒ Cross-validation Folds **10**  
☐ Percentage split % **66**  
 More options...

(Nom) icu

Start Stop

**Result list (right-click for options)**

23:16:28 - trees.J48

**Classifier output**

Time taken to build model: 0.56 seconds

=== Stratified cross-validation ===

=== Summary ===

Correctly Classified Instances	278400	94.2008 %
Incorrectly Classified Instances	17139	5.7992 %
Kappa statistic	0	
Mean absolute error	0.1093	
Root mean squared error	0.2337	
Relative absolute error	99.9973 %	
Root relative squared error	100 %	
Total Number of Instances	295539	

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
1,000	1,000	0,942	1,000	0,970	?	0,500	0,942	0	
0,000	0,000	?	0,000	?	?	0,500	0,058	1	
Weighted Avg.	0,942	0,942	?	0,942	?	?	0,500	0,891	

=== Confusion Matrix ===

a	b	<-- classified as
278400	0	a = 0
17139	0	b = 1

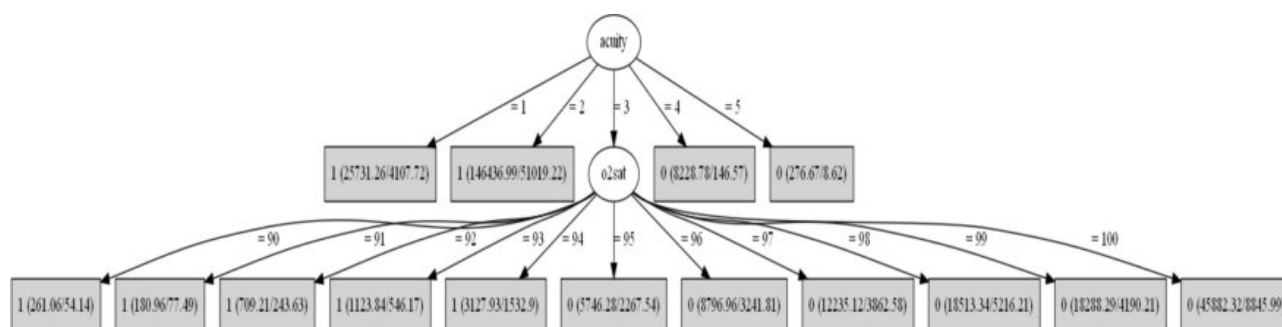
**Status**

OK Log x 0

**Figura 6.5.** Risultati ottenuti dopo l'esecuzione di J48 senza l'uso del filtro "Class Balancer"

### 6.3.1 Stampa dell'albero decisionale

Una volta ottenuti dei risultati che soddisfano i nostri obiettivi non ci rimane che stampare l'albero decisionale prodotto dall'algoritmo di classificazione ed osservarlo.

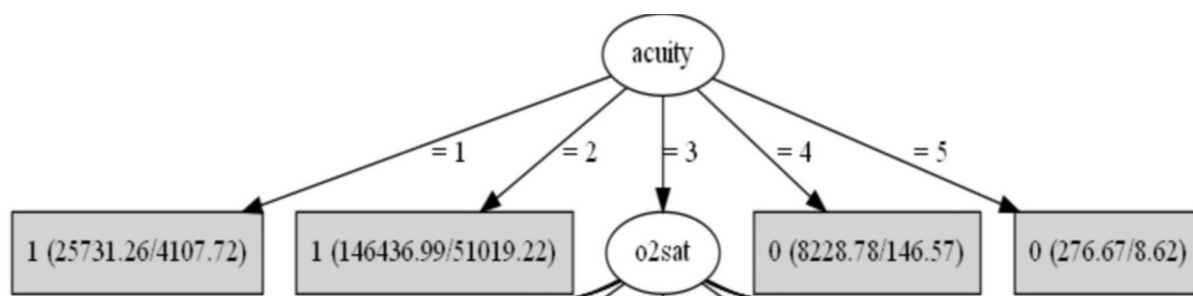


**Figura 6.6.** Albero decisionale prodotto dall'algoritmo di classificazione J48

Come possiamo vedere dall'immagine otteniamo un albero leggibile e molto semplice da interpretare. Sostanzialmente i campi ritenuti "chiave" dal J48 sono "activity" e "o2sat" relativamente, il valore di gravità della situazione assegnato dall'operatore sanitario (in una scala da 5 a 1) al paziente durante il triage seguendo dei parametri ben specifici e il valore di saturazione dell'ossigeno nel corpo.

I numeri che possiamo vedere tra parentesi nei nodi hanno un significato specifico.

Il primo numero è il numero totale di istanze che raggiungono la foglia e il secondo numero è quello delle istanze classificate erroneamente.



**Figura 6.7.** Istanze che raggiungono la foglia/Istanze, di quelle che la raggiungono, classificate erroneamente

L'albero, come già detto, è molto semplice da leggere ed interpretare, quando il campo "activity" ha valore 1 o 2 il paziente viene dato come destinato alla terapia intensiva, quando ha valore 4 o 5 viene classificato come "0" cioè non destinato a finire in terapia intensiva mentre quando assume valore 3 la situazione non è chiara e quindi l'algoritmo ha bisogno di valutare un altro campo per prendere una decisione.

A questo punto entra in gioco nella classificazione il campo "o2sat". Qui anche i dati sono semplici da valutare, semplicemente se il livello di saturazione è  $\leq 94$  allora il paziente viene classificato come indirizzato alla terapia intensiva, altrimenti se  $\text{o2sat} \geq 95$ , viene predetto come "0" cioè non destinato al reparto di terapia intensiva.



## Capitolo 7

# Conclusioni

Dopo aver esaminato i risultati dei nostri esperimenti possiamo quindi fare le dovute considerazioni. Grazie ai nostri studi e agli esperimenti condotti su di loro possiamo affermare che siamo riusciti ad effettuare ed ottenere risultati molto interessanti.

Con la trasformazione da database relazione a Graph DB siamo riusciti ad unire i due progetti MIMIC-IV e MIMIC-ED tutti in un unico grafo così da poter lavorare insieme sui dati di entrambi i progetti, questo allarga di molto la potenzialità delle possibili cose da realizzare su di essi ricordando che MIMIC-ED di fatti integra i dati delle entrate in pronto soccorso con tutti quelli già presenti di MIMIC-IV che riguardano in modo più specifico i ricoveri in terapia intensiva (icu). Quello che è stato fatto è per l'appunto un ottimo esempio di come i dati debbano essere organizzati nel modo giusto ed elaborati per poterli far rendere al massimo, con la nostra conversione ora abbiamo uno strumento molto potente che può gettare le basi per eventuali *sviluppi futuri* come la conversione dei dati raccolti nei due progetti nel modello OMOP [3] partendo già dal Graph DB che, per struttura, gli si avvicina molto.

Anche con il predittore siamo arrivati a dei risultati molto interessanti e abbiamo potuto dimostrare come può essere utile e potente l'utilizzo di algoritmi, applicati nel modo adeguato, nel mondo ospedaliero.

Una volta raccolti i dati vitali di un paziente in entrata al pronto soccorso li abbiamo sottoposti all'algoritmo di classificazione J48 [9] che, configurato in modo adeguato, ritorna un albero decisionale con ottimi risultati di accuratezza. Infatti, le istanze classificate correttamente sono circa il 71 %, il che vuol dire che il nostro predittore riesce a prevedere con successo il futuro clinico di 7 pazienti su 10. Inoltre, in modo in cui lo facciamo è altamente accessibile a tutti, poichè l'albero decisionale prodotto è facile da intendere ed è anche molto rapido da leggere in un contesto dove la tempestività è essenziale.

In eventuali *sviluppi futuri* può essere interessante l'integrazione di altri dati e lo sviluppo di altri predittori che riguardino altri eventi per quanto riguarda il futuro clinico di un paziente una volta entrato in una struttura ospedaliera.





# Bibliografia

- [1] Johnson, Alistair e et al. and PhysioNet. *MIMIC-IV*. 2021. URL: <https://doi.org/10.13026/s6n6-xd98>.
- [2] Johnson et al. *MIMIC-IV-ED*. 2021. URL: <https://doi.org/10.13026/77z6-9w59>.
- [3] OHDSI. *OMOP Common Data Model*. 2022. URL: <https://www.ohdsi.org/data-standardization/the-common-data-model/>.
- [4] Kallfelz, Michael e et al. *MIMIC-IV demo data in the OMOP Common Data Model*. 2021. URL: <https://doi.org/10.13026/p1f5-7x35>.
- [5] Liu Luchen et al. *Learning Hierarchical Representations of Electronic Health Records for Clinical Outcome Prediction*. Mar. 2019. URL: <https://arxiv.org/pdf/1903.08652.pdf>.
- [6] Neo4J. *Graph database concepts*. Gen. 2022. URL: <https://neo4j.com/docs/getting-started/current/graphdb-concepts/>.
- [7] Steven Yang. *How to Map Relational Data to a Graph DB in Four Steps*. Mar. 2018. URL: <https://www.tibco.com/sites/tibco/files/resources/sb-graph-database-final.pdf>.
- [8] Python. *Miscellaneous operating system interfaces*. URL: <https://docs.python.org/3/library/os.html>.
- [9] Nilima Khanna. *J48 Classification (C4.5 Algorithm) in a Nutshell*. Ago. 2021. URL: <https://medium.com/@nilimakhanna1/j48-classification-c4-5-algorithm-in-a-nutshell-24c50d20658e#>.

